Programmieren mit MACRO-SM

Teil I

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens
an der Technischen Universität
Dresden

Die Makroassemblersprache MACRO-SM ist eine moderne flexible Programmiersprache zur Programmierung von Klein- und Mikrorechnern des Systems der Kleinrechentechnik (SKR) der Länder des Rates für Gegenseitige Wirtschaftshilfe (RGW). Während auf den traditionellen Kleinrechenanlagen des SKR die höheren Programmiersprachen wie FORTRAN-77, C, COBOL und PASCAL eine wachsende Bedeutung erfahren haben, hat die Makroassemblersprache MACRO-SM in den letzten Jahren vor allem mit den neuen sowjetischen Mikroprozessorschaltkreisreihen KR 588 und K 1801 für den Einsatz in vielen Geräten und Anlagen der Automatisierungstechnik an Bedeutung gewonnen. Die genannten Mikroprozessorsysteme sind zu den Rechenanlagen SM 4 bzw. Elektronika-60 (M) voll befehlskompatibel.
Mit dieser Artikelreihe soll deshalb der gewachsenen Bedeutung der SKR-Technik Rechnung getragen und dem Leser eine Einführung in die maschinenorientierte Programmierung von Rechenanlagen des SKR und befehlskompatibler Mikroprozessorsysteme gegeben werden.

0. Einleitung

Die Grundmodelle des SKR bilden die Kleinrechnersysteme SM3 und SM4. Sie stellen leistungsfähige Kleinrechnersysteme mit Busstruktur und gleichem Grundbefehlssatz (BIS – Basic Instruction Set) dar, die sowohl für den wissenschaftlich-technischen und ökonomischen Einsatz als auch für den Prozeßrechnereinsatz im Rahmen der Labor- und Prozeßautomatisierung vorgesehen sind. Eine Übersicht über wichtige SKR-Anlagen, die in der DDR im Einsatz sind, ist in Tafel 1 enthalten.
Die Modelle vom Typ SM4 zeichnen sich gegenüber dem Modell SM3 durch eine größere Hauptspeicherkapazität, einen erweiterten Befehlssatz (EIS – Extended Instruction Set) und eine Unterstützung für die Verarbeitung von Gleitkommazahlen (FIS – Floating Instruction Set oder FPP – Floating Point Processor) aus. Bezüglich der Ein- und Ausgabemoduln besteht zwischen allen SKR-Anlagen eine volle Kompatibilität.
Die Kleinrechnersysteme des SKR haben eine modulare Struktur, die flexibel an den jeweiligen Einsatzfall anpaßbar ist. Die Flexibilität begründet sich vor allem auf die Anwendung eines universellen Einheitsbusses mit asynchroner Übertragung und der Möglichkeit des Anschlusses eines breiten Spektrums peripherer Geräte.
Die Kleinrechnersysteme des SKR sind gegenwärtig durch modernere Anlagen auf der Basis von hochintegrierten Schaltkreisen und einer 22-Bit-Hauptspeicherverwaltungseinheit zur Adressierung von max. 4 MByte physischer Hauptspeicherkapazität ergänzt worden.
Darüber hinaus werden die hochintegrierten Schaltkreisfamilien und die genannten Mikroprozessorsysteme verstärkt in der Labor- und Prozeßautomatisierung eingesetzt, wobei vor allem die Makroassemblerprogrammierung Anwendung findet.
In diesem MP-Kurs wird aus den genannten Gründen zuerst eine Einführung in Assemblersprache und Makrotechnik gegeben, und im letzten Teil wird der Maschinenbefehlssatz der SKR-Anlagen behandelt. Die Anwendung wird an einem abschließenden Beispiel demonstriert.

Tafel 1 Übersicht über ausgewählte Rechenanlagen des SKR

Rechenanlage	SM3–10	SM4–10	SM4–20	I100	K1630	I102F	SM14–20	SM52–11 PLUS[6]
Herstellerland	**UdSSR**	UdSSR	ČSSR	SRR	DDR	SRR	UdSSR	ČSSR
Einführungsjahr	1977	1978	1981	1981	1982	1983	1984	1985
max. HS-Größe [KByte]	56	248	248	248	248	248 3840[2]	248 3840[3]	248 3840[4]
Verarbeitungstyp	Parallel, 1 Wort (16 Bit), 1 Byte (8 Bit)							
Befehlslänge	1, 2 und 3 Worte							
Registeranzahl	8	8	8 + 6	8	8(+6)	8 + 6	8 + 6	8 + 6
Befehlssatz	BIS	BIS EIS FIS	BIS EIS FPP	BIS EIS FIS	BIS EIS (FPP)[1]	BIS EIS FPP	BIS EIS FPP	BIS EIS FPP[5]
Befehlsausführungszeit für Reg.-Reg.-Befehle [µs]	5,0	1,2	2,3	2,3	3,5–4,2	0,5	1,0	0,34

[1] Wahlweise mit FPP; [2] Wahlweise mit 22-Bit-Hauptspeicherverwaltung; [3] Seit 1985 mit 22-Bit-Hauptspeicherverwaltung; [4] Seit 1986 mit 22-Bit-Hauptspeicherverwaltung; [5] FPP-Befehle zusätzlich auch durch das Mikroprogramm realisiert; [6] Wahlweise mit freier Mikroprogrammierung

1. Einführung in die Makroassemblersprache

MACRO-SM ist eine moderne Assemblersprache, die sich durch Flexibilität, Transparenz und ein entwickeltes Makrokonzept auszeichnet. Die Übersetzung der Programme erfolgt im allgemeinen in eine Objektsprache. Ein Programm in der Objektsprache wird als Objektmodul bezeichnet, ist verschieblich und kann vom Taskbuilder in eine absolute, nicht verschiebliche, abarbeitungsfähige Task (Lademodul) umgewandelt werden. Der Taskbuilder unterstützt dabei die Verbindung einer Vielzahl von Objektmoduln zu einer Task, wobei infolge der Realisierung globaler Symbole in der Makroassemblersprache zwischen den Objektmoduln Querbezüge hergestellt werden können. Durch diese Eigenschaften wird insbesondere eine modulare Programmierung unterstützt, die eine bessere Ausnutzung der Ressourcen gestattet sowie die Transparenz der Programmsysteme und die Wiederverwendbarkeit und Änderungsfreundlichkeit der einzelnen Programmbausteine (Moduln) erhöht. Für spezielle Anwendungsfälle besteht die Möglichkeit der Übersetzung in einen absoluten Lademodul. Die Realisierung umfangreicher Möglichkeiten der bedingten Übersetzung gestattet die Generierung von an den jeweiligen Anwendungsfall angepaßten Programmsystemen aus einem Quellmodul. Das ausgereifte Makrokonzept stellt ein effektives Rationalisierungsmittel für die Programmentwicklung dar. Es besteht neben der Verwendung von programmspezifischen Makrodefinitionen die Möglichkeit der Anwendung einer umfangreichen Systemmakrobibliothek und nutzereigener Privatbibliotheken. Die hohe Flexibilität des Assemblers drückt sich außerdem im Vorhandensein umfangreicher Möglichkeiten zur Steuerung des Übersetzungsprozesses und der Protokollgestaltung aus.

1.1. Das Format der Anweisungen

Eine Anweisung kann im freien Format aus

maximal vier Feldern bestehen:

name: operation operand(en) ; kommentar

Eine *Anweisung* wird immer als Quellzeile betrachtet und in einer Druckzeile protokolliert. Die maximale Länge einer Zeile darf 132 Zeichen betragen. Es wird aber auf Grund von Einschränkungen bei verschiedenen E/A-Geräten empfohlen, die Zeilenlänge von 80 Zeichen nicht zu überschreiten. Jedes der vier Felder einer Anweisung kann bei der Programmierung ausgelassen werden. Wenn alle vier Felder ausgelassen werden, entsteht eine Leerzeile (Leeranweisung). Leerzeilen sind im Programm zulässig. Eine Anweisung wird vom Makroassembler von links nach rechts abgearbeitet. Wird im Namensfeld ein Symbol angegeben, so wird dem Symbol der augenblickliche Wert des Speicherplatzzuordnungszählers (SZZ) zugewiesen.
Das *Namensfeld* wird als ein solches durch den Doppelpunkt (:) erkannt, der dem Symbol folgen muß. Im Namensfeld dürfen auch mehrere Symbole definiert werden, wobei jedes Symbol durch einen Doppelpunkt abgeschlossen werden muß und jedem Symbol der gleiche Wert zugewiesen wird.

Beispiele:
```
START:
ANFANG: MOV R0,-(SP) ; RETTEN R0
A0:B0: FELD:  .BLKW 20
```

Das Operationsfeld folgt dem Namensfeld. Bei fehlendem Namensfeld beginnt die Anweisung mit dem Operationsfeld. Im Operationsfeld steht ein Symbol, das die mnemonische Bezeichnung des Maschinenbefehls, der Assembleranweisung oder des Makrorufes festlegt. Begrenzer des Operationsfeldes sind Leerzeichen (SP), Tabulator (HT) und alle Sonderzeichen, die nicht im Radix-50-Kode (Tafel 2) enthalten sind.

Beispiele:
```
ZYKL: MOV(R0)+,R1 ;LADEN VON
A(I+1) MOV R1,A
       MOV #25,R4
```

Bei ausgelassenem Operationsfeld wird die Anweisung als eine **.WORD**-Anweisung interpretiert. Ein ausgelassenes Operationsfeld wird auch angenommen, wenn das Symbol im Operationsfeld nicht definiert ist, z. B. bei einem fehlerhaften Operationskode.
Das *Operandenfeld* spezifiziert nach dem Operationsfeld die Operanden, mit denen die Operation ausgeführt werden soll. Werden mehrere Operanden angegeben, so müssen sie untereinander durch Komma (,), Tabulator (HT) oder Leerzeichen (SP) getrennt werden.

Beispiele:
```
M25:
PUT #FELD,#80.,#40 ; AUSGABE
MOV A R0
```

Das Operandenfeld endet mit dem Semikolon (;), das einen Kommentar einleitet, oder bei fehlendem Kommentar am Zeilenende.

Tafel 2 RADIX-50-KODE

Zeichen	Erstes Kode-zeichen	Zweites Kode-zeichen	Drittes Kode-zeichen
SP	000000	000000	000000
A	003100	000050	000001
B	006200	000120	000002
C	011300	000170	000003
D	014400	000240	000004
E	017500	000310	000005
F	022600	000360	000006
G	025700	000430	000007
H	031000	000500	000010
I	034100	000550	000011
J	037200	000620	000012
K	042300	000670	000013
L	045400	000740	000014
M	050500	001010	000015
N	053600	001060	000016
O	056700	001130	000017
P	062000	001200	000020
Q	065100	001250	000021
R	070200	001320	000022
S	073300	001370	000023
T	076400	001440	000024
U	101500	001510	000025
V	104600	001560	000026
W	107700	001630	000027
X	113000	001700	000030
Y	116100	001750	000031
Z	121200	002020	000032
Sputnik	124300	002070	000033
.	127400	002140	000034
unbenannt	132500	002210	000035
0	135600	002260	000036
1	140700	002330	000037
2	144000	002400	000040
3	147100	002450	000041
4	152200	002520	000042
5	155300	002570	000043
6	160400	002640	000044
7	163500	002710	000045
8	166600	002760	000046
9	171700	003030	000047

Anmerkung:
Die Umwandlung von drei ASCII-Zeichen $\alpha_1\alpha_2\alpha_3$ erfolgt nach der Formel $\alpha_1 \cdot 50^2 + \alpha_2 \cdot 50 + \alpha$.
Beispiel: ZA5 = 121200 + 50 + 43 = 121313

Das *Kommentarfeld* kann alle druckbaren Zeichen des ASCII- oder GOST-Kodes enthalten und dient der Erläuterung der Anweisungen des Programms. Es hat keinen Einfluß auf den Übersetzungsvorgang und auf den Objektmodul. Fehlen die anderen Felder der Anweisung, so kann das Kommentarfeld ab Position 1 beginnen (Kommentaranweisung).

Beispiele:
```
START: START DES HAUPTPRO-
       ;GRAMMES FUER DIE
       ;VERSUCHSAUSWERTUNG
;BENUTZTE UNTERPROGRAMME:
;EAL, KONVD, QUAD
```

Anmerkungen:
1. Zur übersichtlichen Gestaltung der Druckprotokolle können beliebig viele Leerzeichen (SP) und/oder Tabulatoren (HT) zwischen den Feldern, Operanden bzw. Symbolen und Termen eines Ausdruckes eingefügt werden.
2. Zur einheitlichen Gestaltung der Druckprotokolle wird empfohlen, das Namensfeld ab Position 1, das Operationsfeld ab Position 9, das Operandenfeld ab Position 17 und das Kommentarfeld ab Position 33 zu beginnen. Diese Positionen werden automatisch eingestellt, wenn die Felder mit einem Tabulator (HT) (zusätzlich) begrenzt werden, da ein Tabulator immer einen Sprung auf die nächste durch 8 teilbare Position plus 1 bewirkt (9, 17, 25, 33, 41 usw.). Vor dem Kommentarfeld sind eventuell bei fehlendem oder kurzem Operandenfeld zwei Tabulatoren erforderlich.

1.2. Das Prinzip der Übersetzung

Die Übersetzung von MACRO-Quellprogrammen erfolgt nach dem Prinzip eines Zwei-Pass-Assemblers. Im ersten Assemblerpass werden die internen Datenbereiche wie dynamischer Speicherbereich, Pufferbereiche und Filesteuerbereiche initialisiert. Anschließend erfolgt die Eingabe und Analyse der Kommandozeile, die die Filespezifikationen für das Quellfile, Makrobibliotheksfile, Objektfile und Listenfile sowie weitere Steuerinformationen enthält. Nach diesen Vorbereitungen erfolgt das Einlesen der Quellzeilen und ihre Analyse.
Das Hauptziel des ersten Passes besteht im Erstellen der Nutzersymboltabelle (UST) und der Makrosymboltabelle (MST). Zu diesem Zweck muß die Makrogenerierung durchgeführt und jeder Maschinenbefehl soweit übersetzt werden, daß seine Länge bestimmt werden kann (1, 2 oder 3 Worte). Durch Addieren der Befehls- und Speicherbereichslängen wird der Speicherplatzzuordnungszähler (SZZ) gebildet, der zum Aufbauen der UST benötigt wird. Gleichzeitig wird im ersten Pass die Subtiteltabelle in das Listenfile ausgegeben.
Im zweiten Pass werden die Quellzeilen noch einmal eingelesen und im Prinzip die gleichen Arbeitsschritte wie im ersten Pass ausgeführt, mit dem Ziel der vollständigen Übersetzung der Maschinenbefehle unter Verwendung der im ersten Pass aufgebauten Symboltabelle. Es werden dabei gleichzeitig der Objektmodul und das Assemblerprotokoll (Listfile) erzeugt. Nach der Übersetzung werden die Symboltabelle und gegebenenfalls die Cross-Reference-Tabelle in das Listenfile ausgegeben.

1.3. Alphabet

Das *Alphabet* von MACRO-SM umfaßt:
- die Großbuchstaben des lateinischen Alphabets von A bis Z
- die arabischen Ziffern von 0 bis 9
- Sonderzeichen . ¤ : = % # @ ( ) , ; < > + − * / & ! “ ‘ ^ ?
- Steuerzeichen FF HT SP.

Zusammenfassend ist die Bedeutung der einzelnen Zeichen in Tafel 3 dargestellt. Außer diesen Zeichen sind in Kommentaren alle anderen druckbaren Zeichen zulässig. Die Verwendung der kleinen lateinischen Buchstaben des ASCII-Kodes bzw. der Großbuchstaben des kyrillischen Alphabets des GOST-Kodes ist bei **.ENABL LC** möglich.

1.4. Symbole

Von der programmtechnischen Realisierung her werden die Symbole in permanente Symbole, Nutzersymbole, Makrosymbole und lokale Symbole unterschieden.



Kurs

Permanente Symbole dienen zur Bezeichnung der Maschinenbefehle und Assembleranweisungen. Sie sind in der Tabelle der permanenten Symbole (PST) zusammengefaßt. *Nutzersymbole* sind vom Nutzer im Programm definierte Symbole, denen entweder der augenblickliche Wert des SZZ (:) oder ein beliebiger anderer Wert durch die Direktanweisung (=) zugewiesen wird. Nutzersymbole werden im ersten Assemblerpass in die Nutzersymboltabelle (UST) eingetragen. Die UST wird nach der Übersetzung gedruckt (außer bei **.NLST SYM**).
Makrosymbole sind die symbolischen Namen von Makrodefinitionen, die entweder im Programm definiert (**.MACRO**) oder aus einer Makrobibliothek aufgerufen (**.MCALL**) werden können. Sie werden ebenfalls im ersten Pass in die Makrosymboltabelle (MST) eingetragen.
Für die Bestimmung eines Symbols ist die Reihenfolge des Suchens in den Tabellen entscheidend. Die Symbole des Operationsfeldes werden wie folgt in den Tabellen gesucht: MST, PST, UST.
Lokale Symbole sind spezielle numerische Symbole (z. B. **1⊠**, **2⊠** usw.) mit einem eingeschränkten Gültigkeitsbereich, die ähnlich wie Nutzersymbole verwendet werden. Besonders vorteilhaft werden sie für interne Sprungmarken verwendet, wenn dem Sprungziel keine mnemonische Bezeichnung zugeordnet werden braucht. Lokale Symbole werden in lokale Symbolblöcke (LSB) eingetragen. Auf Grund des eingeschränkten Gültigkeitsbereichs können automatisch oder nutzergesteuert mehrere LSB nacheinander eingerichtet werden. Ein lokales Symbol belegt nur 3 Worte im LSB, während ein Symbol in der UST 4 Worte belegt.

1.4.1. Nutzer- und Makrosymbole

Nutzer- und *Makrosymbole*, wie auch permanente Symbole, werden nach gleichen syntaktischen Regeln gebildet. Da diese Symbole intern im Assembler im Radix-50-Kode dargestellt werden, sind für die Bildung von Symbolen alle Radix-50-Zeichen zulässig:

- Großbuchstaben des lateinischen Alphabets
- arabische Ziffern
- Sonderzeichen **.** und ⊠

Die Sonderzeichen **.** und ⊠ werden als Alphazeichen gewertet und sind für die Bildung von Systemsymbolen reserviert. Zur Vermeidung von Konflikten sollten diese Zeichen vom Anwendungsprogrammierer nicht benutzt werden.
Ein Symbol besteht aus 1 bis 6 Zeichen, wobei das erste Zeichen immer ein Alphazeichen sein muß. Längere Symbole sind zulässig, es sind aber nur die ersten 6 Zeichen signifikant.

Beispiele:
A, R0, A250WZ, ALPHA, ALPHA250, ALPHA251
.WRITE, ⊠⊠⊠SYM, .WORD, ⊠.25., A⊠DEZ

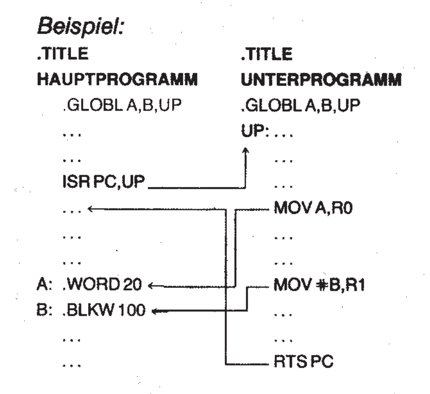
Die Symbole ALPHA250 und ALPHA251 sind somit identisch und entsprechen dem Symbol AlPHA2.
Hinweis:
Ein Symbol wird durch Leerzeichen (SP) und jedes nicht im Radix-50-Kode definierte Symbol begrenzt.
Der Gültigkeitsbereich der Nutzer- und Makrosymbole erstreckt sich über das gesamte Programm, unabhängig von einer Aufteilung in Programmabschnitte. Diese Symbole werden deshalb als interne Symbole eines Programms (einer Übersetzungseinheit) bezeichnet.
Die Herstellung von Querbezügen zwischen getrennt übersetzten Quellprogrammen ist mit dem Taskbuilder unter Verwendung globaler Symbole möglich. Man unterscheidet dabei zwischen *globalen Definitionen* und *globalen Bezugnahmen.* Ein Symbol ist global definiert, wenn es im Programm *global-definiert* ist (:: bzw. ==) oder durch eine **.GLOBL**-Anweisung als globales spezifiziert wird.

Tafel 3 Alphabet der Assemblersprache MACRO-SM

Zeichen	Bedeutung/Funktion
A–Z	Alphazeichen (Großbuchstaben des lateinischen Alphabets)
⊠ ($)	Zusätzliches Alphazeichen (reserviert für Systemsymbole); Kennzeichen für lokale Symbole
.	Zusätzliches Alphazeichen in Verbindung mit anderen Alphazeichen (reserviert für Systemsymbole); Speicherplatzzuordnungszähler (SZZ) des Assemblers; Dezimalpunkt nach Zahlen
0–9	Numerische Zeichen
:	Symboldefinition, Begrenzer des Namensfeldes
=	Wertzuweisung für ein Symbol oder SZZ
%	Registerkennzeichen
#	Direktwertkennzeichen
@	Kennzeichen der Speicherindirektadressierung
()	Registerindirektadressierung
,	Trennzeichen für Operanden
;	Kommentarkennzeichen
<>	Klammerung für Argumente und Ausdrücke
+	Additionszeichen und Autoinkrementkennzeichen
–	Subtraktionszeichen und Autodekrementkennzeichen
*	Multiplikationszeichen
/	Divisionszeichen
&	Logisches UND
!	Logisches (inklusives) ODER
"	Kennzeichen für die Definition von zwei ASCII-Zeichen
'	Kennzeichen für die Definition von einem ASCII-Zeichen; Begrenzer für formale Argumente in Makrodefinitionen
^	Universeller einstelliger Operator; Kennzeichen der Pfeilkonstruktion
\	Operator für numerische Symbole im Makroaufruf
?	Kennzeichen für die automatische Generierung eines lokalen Symbols für ein formales Argument, falls keine Substitution durch einen aktuellen Parameter erfolgt
SP	Begrenzer eines einzelnen Symbols oder Feldes
HT	Begrenzer eines einzelnen Symbols oder Feldes
FF	Seitenwechsel

Ein Symbol realisiert eine globale Bezugnahme, wenn es im Programm nicht definiert ist (bei **.ENABL GBL**) bzw. durch eine **.GLOBL**-Anweisung als *global* spezifiziert wird (bei **.DSABL GBL**). In beiden Fällen bekommt das Symbol das Attribut *global.*

Beispiel:



Nutzersymbole werden außerdem durch ein *Verschieblichkeitsattribut* charakterisiert. Nichtverschiebliche Symbole haben das Attribut *„absolut“*, verschiebliche das Attribut *„relativ“*. In nichtverschieblichen *Programmabschnitten* (PA) dürfen nur absolute Symbole verwendet werden. In verschieblichen PA dürfen relative und absolute Symbole verwendet werden.
Wird ein Symbol im Namensfeld einer Anweisung definiert (:), dann werden ihm immer der Wert und das Verschieblichkeitsattribut des SZZ zugewiesen. Der SZZ wird in nichtverschieblichen PA immer in absoluten Adressen und in verschieblichen PA immer in relativen Adressen geführt (entspricht dem Verschieblichkeitsattribut des PA).
Ein Symbol, das durch eine Direktanweisung definiert wird, erhält immer das Verschieblichkeitsattribut des Ausdrucks rechts des Gleichheitszeichens (=).
Registersymbole sind die standardisierten Bezeichnungen für die allgemeinen Prozessorregister. Die Registersymbole sind im Assembler permanent definiert.

1.4.2. Lokale Symbole

Lokale Symbole werden aus einer natürlichen Dezimalzahl und dem Sonderzeichen ⊠ gebildet. Die Dezimalzahl darf den Wert von **1** bis **65535** annehmen. Z. B. **1⊠**, **2⊠**, ... **65535⊠**.
Die Symbole **64⊠** bis **127⊠** sind für die Makrogenerierung reserviert.
Der *Gültigkeitsbereich* lokaler Symbole erstreckt sich standardgemäß (**.DSABL LSB**) auf den Bereich zwischen zwei Anweisungen mit einem Symbol im Namensfeld. Für diesen Bereich wird ein lokaler Symbolblock (LSB) aufgebaut, wenn wenigstens ein lokales Symbol definiert wird. Außerdem wird mit jeder **.PSECT**-Anweisung ein neuer LSB eröffnet.
Zur Erhöhung der Effektivität des Assemblers kann der Programmierer den Gültigkeitsbereich der lokalen Symbole steuern und die Kapazität der LSB besser ausnutzen. Ein LSB, der mit einer **.ENABL-LSB**-Anweisung eingerichtet wurde, besitzt seine Gültigkeit bis zur nächsten **.ENABL-LSB-**, **.DSABL-LSB-** oder **.PSECT**-Anweisung.

Das Verschieblichkeitsattribut lokaler Symbole entspricht immer dem Verschieblichkeitsattribut des PA.

1.5. Konstanten

In den Ausdrücken im Operandenfeld der Anweisungen können *Konstanten* benutzt werden, die als Zahlen, Zeichenkonstanten oder Zeichenkettenkonstanten definiert sein können. Alle Konstanten werden intern in eine 16-Bit-Zahl umgerechnet.
Numerische Zeichenketten ohne Angabe der Basis des Zahlensystems werden entsprechend der aktuellen Basis des Zahlensystems interpretiert. Die Basis des Zahlensystems wird durch die **.RADIX**-Anweisung festgelegt. Bei ausgelassener **.RADIX**-Anweisung wird die Basis 8 angenommen. Mit dem universellen einstelligen Operator **^x**, besteht die Möglichkeit, die Basis des Zahlensystems temporär für einen Wert zu ändern:

^B – Binärzahlen: **^B101110,^B100**
^O – Oktalzahlen: **^056,^0177777**
^D – Dezimalzahlen: **^D99,^D32767**

Für den Operator **^D** existiert eine Kurzschreibweise mit nachgestelltem Dezimalpunkt, z. B. **99., 32767.**
Als zusätzliche Operatoren sind die Vorzeichen- und Komplementoperatoren zulässig:
^C – Bildung des Einerkomplements:
^C^B100,^C^056
– – Bildung des Zweierkomplements:
–^B100,–99.,^D–99

Der Operator ± als Vorzeichenoperator ist zulässig, aber redundant.
Zeichen- und Zeichenkettenkonstanten werden mittels der Operatoren **'** und **"** von druckbaren Zeichen des GOST- bzw. ASCII-Kodes gebildet. Der Operator **'** bewirkt, daß das nachfolgende ASCII-Zeichen in den 7-Bit-Kode umgewandelt und als 16-Bit-Wert dargestellt wird:
'A ≙ ^0101,'+ ≙ ^053 usw.

Der Operator **"** bewirkt, daß die zwei nachfolgenden ASCII-Zeichen entsprechend der 7-Bit-Kodetabelle in einen 16-Bit-Wert umgewandelt werden, so daß das erste Zeichen im niederwertigen Byte und das zweite im höherwertigen Byte gespeichert ist:
"AB ≙ ^0041101,"+3 ≙ ^0031453

1.6. Terme und Ausdrücke

Unter einem *Term* versteht man die elementaren syntaktischen Einheiten eines Ausdrukkes. Terme sind:

– Symbole (Nutzer- und permanente Symbole), z. B. **FELD, A0, MARK**
– lokale Symbole, z. B. **54⊠, 23⊠**
– der Speicherplatzzuordnungszähler (SZZ).

Eine Bezugnahme auf den aktuellen Wert des SZZ erfolgt durch einen alleinstehenden Punkt (.), z. B. **.+20, A0–.+2**
– Konstanten (Zahlen mit und ohne Angabe der Basis des Zahlensystems und Zeichenkettenkonstanten), z. B. **^B101, 25, 25., 'A**
– Terme, auf die ein einstelliger Operator angewendet wird, z. B. **^C25, +25, –25., ^C"AB, ^C+25.**
– Terme, die durch spitze Klammern oder die Pfeilkonstruktion geklammert sind, z. B.
<**+25**>, <**^C"AB**>, ^/**^C"AB**/, ^#^**C**<**+25.**>#

Die Pfeilkonstruktion **^x** . . . *x*, wobei *x* ein beliebiges Zeichen ist, wird analog den spitzen Klammern zur Klammerung von Termen, Ausdrücken und Argumenten benutzt. Das zur Klammerung benutzte Zeichen *x* darf innerhalb der Klammern nicht nochmals verwendet werden. Die Pfeilkonstruktion wird insbesondere in der Makrotechnik angewendet, wo die spitzen Klammern unzulässig oder unübersichtlich sind.

– Ausdrücke, die durch spitze Klammern oder die Pfeilkonstruktion geklammert sind, z. B.
<A0–B>, <'Z–'A–1>, ^Z BETA–25 Z

Die Anwendung mehrerer einstelliger Operatoren auf einen Term ist zulässig. Die Abarbeitung der einstelligen Operatoren erfolgt in der Reihenfolge von rechts nach links.

Beispiel:
Der Term **+–^C^0177776**
entspricht dem Term
<+<–<**^C**<**^0177776**>>>>.

Unter einem *Ausdruck* versteht man einen Term oder mehrere Terme, die untereinander durch zweistellige Operatoren verknüpft sind. Zweistellige Operatoren sind:

+ Addition
– Subtraktion
***** Multiplikation
/ Division
& logisches UND
! logisches (inklusives) ODER.

Alle zweistelligen Operatoren haben die gleiche Priorität. Die Berechnung der Ausdrücke erfolgt so, daß zuerst alle Terme berechnet werden und anschließend die Werte der Terme durch die zweistelligen Operatoren in der Reihenfolge von links nach rechts verknüpft werden.

Beispiel:

```
   P2 - P1 / 16. * 16. + <FELD ! ^C^ 017>
1. |       |   |     |     |      | |__| |
2. |       |   |     |     |      |____| |
3. |       |   |     |     |___________| |
4. |_______|   |     |                   |
5. |___________|     |                   |
6. |_________________|                   |
7. |_____________________________________|
```

Bei der Ausdrucksberechnung wird für nichtdefinierte Symbole oder fehlerhafte Terme der Wert 0 angenommen. Steht zwischen zwei Termen kein zweistelliger Operator, so werden die Terme links und rechts des Trennzeichens als getrennte Ausdrücke gewertet.

Beispiel:
A+2 MASK1&MASK2

Da zwischen den Termen **2** und **MASK1** kein zweistelliger Operator steht, enthält das Beispiel zwei Ausdrücke, **A+2** und **MASK1&MASK2**.

Die Ergebnisse und Zwischenergebnisse der Ausdrucksberechnung werden intern als 16-Bit-Werte dargestellt. Bei Überschreitung der Stellenzahl werden die Werte linksseitig beschnitten (T-Fehler). Steht ein Ausdruck in einer **.BYTE**-Anweisung, so wird das Ergebnis nach der Ausdrucksberechnung auf 8 Bit verkürzt.
Bei der Ausdrucksberechnung wird das *Verschieblichkeitsattribut* des Ergebnisses ermittelt. Das Ergebnis eines Ausdruckes darf in einem verschieblichen PA absolut oder relativ sein. Ein Ausdruck ist absolut, wenn die Anzahl der relativen Terme mit positiven Vorzeichen gleich der Anzahl der relativen Terme mit negativem Vorzeichen ist.
Ein Ausdruck ist relativ, wenn die Anzahl der Terme mit positivem Vorzeichen um 1 größer ist als die Anzahl der Terme mit negativem Vorzeichen.
Die Multiplikation und Division relativer Terme sowie logische Operationen mit relativen Termen sind unzulässig.

Beispiel:
Es sei angenommen,
daß die Symbole **A, B, C** und **D** relativ und **I, J, K** und **L** absolut sind.
A–B absolut
A–B–C falsch (negativ verschieblich!)
A–B+C relativ
C–D+A+B falsch (doppelt verschieblich!)
I+2–A+B absolut
4*C–D falsch (Multiplikation!)
4*<C–D> absolut
C–D/16*16+<K!L> absolut
A+<I&J&<K!L>> relativ

Sind in einem Ausdruck globale Bezugnahmen oder Bezugnahmen auf Symbole anderer PA vorhanden, so wird die Ausdrucksberechnung teilweise oder vollständig in den Taskbuilder verlagert, wo die Ausdrucksberechnung zur Taskbuilderzeit nach den gleichen Regeln erfolgt. Das Verschieblichkeitsattribut einer globalen Bezugnahme entspricht dem globaldefinierten Symbol im anderen Quellprogramm.

wird fortgesetzt

Literatur

/1/ Merkel, G.: Neue Gerätesysteme des ESER und SKR. rechentechnik/datenverarbeitung, Berlin 16, (1979) Beiheft, S. 2–8
/2/ Programmnoje obespetschenije SM–3. Diskovaja operazionnaja sistema obschtschewo nasnatschenija. Programma Makroassembler. 4.072.088 IE2
/3/ Horn, Th.; Kofer, C.: Handbuch der SKR-Programmierung. Schriftenreihe „Informationsverarbeitung im Hoch- und Fachschulwesen", Heft 1. ZLO Zwickau (1979)
/4/ Horn, Th.; Utke, M.: Programmierung in der Assemblersprache MACRO-SM. Heft 3 der Schriftenreihe „Programmierung von Rechenanlagen des SKR".
Ingenieurhochschule Dresden (1984)

Programmieren mit MACRO-SM

Teil II

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens
an der Technischen Universität Dresden

1.7. Direktanweisung

Mit der *Direktanweisung* kann man Symbolen einen Wert zuweisen. Sie hat folgende Syntax:

name: *symbol* = *expr* ; *kommentar*

Das Namensfeld und das Kommentarfeld können auch entfallen. Dem Symbol *symbol* wird der Wert des Ausdruckes *expr* zugewiesen. Bei der ersten Wertzuweisung wird das Symbol in die UST eingetragen. Bei weiteren Wertzuweisungen wird nur der Wert des Symbols in der UST geändert.
Das Symbol kann mit == bzw. durch eine **.GLOBL**-Anweisung als global erklärt werden. Eine Bezugnahme auf globale nichtdefinierte Symbole ist nicht erlaubt. Bei der Berechnung des Ausdruckes sind einstufige Vorwärtsbezugnahmen auf Symbole zulässig. Das Verschieblichkeitsattribut des Symbols wird durch den Ausdruck bestimmt. In verschieblichen PA darf der Wert des Ausdrucks relativ oder absolut sein.

Beispiel:
```
ALPHA = BETA + 10
          ;VORWAERTSBEZUGNAHME
          ;AUF BETA
BETA = 100
          ;ZUWEISUNG EINES
          ;ABSOLUTEN WERTES
FELDN = FELD + N - 1
          ;ZUWEISUNG EINES
          ;RELATIVEN WERTES
```

1.8. Speicherplatzzuordnungszähler

Der Speicherplatzzuordnungszähler (SZZ) wird durch einen alleinstehenden Punkt (.) bezeichnet. In Verbindung mit Radix-50-Zeichen hat der Punkt die Bedeutung eines Alphazeichens oder eines Dezimalpunktes! Wird der SZZ im Operandenfeld verwendet, so entspricht sein Wert der Adresse des ersten Wortes des Maschinenbefehls. Bei Assembleranweisungen entspricht der Wert des SZZ der Adresse des aktuellen Wortes oder Bytes.

Beispiel:
```
M0: MOV .,R2
;DER SZZ ENTSPRICHT DEM SYMBOL
;M0
FELD: .WORD 20,ALPHA,.,20.
;SZZ ENTSPRICHT DER ADRESSE
;FELD + 4
```

Das Verschieblichkeitsattribut des SZZ wird durch den Charakter des PA bestimmt. Für nichtverschiebliche PA ist der SZZ absolut und für verschiebliche PA relativ. Am Anfang eines PA bzw. der Übersetzung wird dem SZZ immer der Wert 0 zugewiesen. Mit der Direktanweisung kann dem SZZ ein beliebiger Wert zugewiesen werden. Die Direktanweisung kann auch zum Reservieren von Speicherbereichen benutzt werden.

Beispiel:
```
. = 17500      ;DER SZZ WIRD AUF 17500
               ;GESETZT
. = . + 120    ;DER SZZ WIRD UM 120
               ;BYTES
               ;WEITER GERUECKT
```
Die letzte Schreibweise ist identisch mit der Anweisung
```
.BLKB 120      ;RESERVIERE 120 BYTES
```

2. Allgemeine Assembleranweisungen

In diesem Abschnitt werden die wesentlichen Eigenschaften der allgemeinen Assembleranweisungen beschrieben. Das Format der Assembleranweisungen entspricht den Festlegungen gemäß 1.1, wobei auch für Assembleranweisungen in der Regel ein Symbol im Namensfeld zulässig ist. Da die Verwendung von Symbolen im Namensfeld von Assembleranweisungen (mit Ausnahme der Definitionsanweisungen) nicht von grundsätzlicher Bedeutung ist, wird darauf im weiteren nicht besonders hingewiesen.

2.1. Anweisungen zur Auswahl von Assemblerfunktionen

Bestimmte Funktionen des Makroassemblers lassen sich mit der Anweisung
.ENABL *arg*
einschalten bzw. mit der Anweisung
.DSABL *arg*
ausschalten, wobei *arg* die Funktion des Makroassemblers entsprechend Tafel 4 spezifiziert.
Die Funktionsauswahl kann auch mit den Schaltern /**EN:***arg* und /**DS:***arg* in der Kommandozeile erfolgen. Die durch die Schalter beeinflußten Argumente haben gegenüber den Argumenten der Anweisungen den Vorrang.

2.2. Anweisungen zur Steuerung des Übersetzungsprotokolls

Das Übersetzungsprotokoll enthält standardgemäß für jede Anweisung ein Fehlerkennzeichenfeld, in dem maximal vier Fehler angezeigt werden, den Speicherplatzzuordnungszähler (SZZ), eine dezimale Nummer der eingelesenen Quellanweisung, eine Spalte für den Objektkode in der maximalen Länge von 3 Worten und eine Spalte für die vollständige Quellanweisung. Die Kopfzeilen eines jeden Blattes enthalten den Programm-

Tafel 4 Steuerfunktionen des Assemblers

Argument	Standard	Funktion
ABS	disable	Ausgabe des absoluten Maschinenkodes im OS-RW-Format. Konvertieren in das formatierte Binärformat (für den Absolutlader) des Dienstprogramms FLX.
AMA	disable	Die relative Adressierung (Modifikation 67 gemäß 5.3) wird bei der Assemblierung durch die absolute Adressierung ersetzt (Modifikation 37). Diese Funktion erleichtert den Programmtest.
CDR	disable	Die Zeichen ab Position 73 werden als Kommentar gewertet. Damit können die Positionen 73–80 bei einer Lochkarteneingabe als Problem-Folgefeld benutzt werden.
CRF	disable	Ausgabe einer Cross-Referenz-Liste. Bei **.ENABL CRF** wird eine Cross-Referenz-Liste erstellt.
FPT	disable	Gleitkommazahlen werden nicht gerundet (Truncation). Bei **.DSABL FPT** werden die Gleitkommazahlen gerundet.
GBL	enable	Alle undefinierten symbolischen Bezugnahmen (Referenzen) werden als global-undefinierte Bezugnahmen angenommen, für die zur Taskbuilderzeit eine Auflösung erwartet wird (Bezugnahme auf globale Symbole anderer Objektmodule). Bei **.DSABL GBL** führen alle offenen symbolischen Bezugnahmen zu undefinierten Symbolen, die mit einem U-Fehler im Assemblerprotokoll angezeigt werden.
LC	disable	Die Kleinbuchstaben des ASCII-Kodes werden bei der Eingabe des Quellfiles nicht in Großbuchstaben umgewandelt. Im GOST-Kode ist somit auch eine Verarbeitung von kyrillischen Schriftzeichen möglich. Bei **.DSABL LC** erfolgt eine Umkodierung der kleinen bzw. kyrillischen Schriftzeichen in große lateinische.
LSB	disable	Lokale Symbolblöcke werden gewöhnlich durch ein Symbol im Namensfeld oder durch eine **.PSECT**-Anweisung begrenzt. Bei **.ENABL LSB** wird ein lokaler Symbolblock aufgebaut, dessen Gültigkeit sich bis zur nächsten **.DSABL LSB**-, **.ENABL LSB**- oder **.PSECT**-Anweisung erstreckt.
PNC	disable	Ausgabe des Binärkodes. Bei **.DSABL PNC** wird die Ausgabe des Binärkodes (Maschinenkodes) unterdrückt.
REG	enable	Die Standard-Registersymboldefinitionen werden verwendet (**R0 = %0, R1 = %1, .., R5 = %5, SP = %6, PC = %7**). Bei **.DSABL REG** werden keine Standard-Registersymboldefinitionen vom Assembler durchgeführt.

```
PSECT - BEISPIELE          MACRO M28     Ø4-JUL-84 17:25  PAGE 1


     1                                .TITLE  PSECT - BEISPIELE
     2 ØØØØØØ                         .PSECT  PA1,ABS
     3 ØØØØØØ   Ø127ØØ   ST:          MOV     #1ØØ.,RØ
                ØØØ144
     4 ØØØØØ4   Ø127Ø1                MOV     #BER,R1
                ØØØØØØ'
     5 ØØØØ1Ø   ØØ5Ø21   1¤:          CLR     (R1)+
     6 ØØØØ12   ØØ53ØØ                DEC     RØ
     7 ØØØØ14   ØØ1375                BNE     1¤
     8 ØØØØØØ                         .PSECT
     9 ØØØØØØ            BER:         .BLKW   1ØØ.
    1Ø ØØØ31Ø            CØ:          .BLKW   4
    11 ØØØ32Ø            C1:          .BLKW   2
    12 ØØØØ16                         .PSECT  PA1
    13 ØØØØ16   Ø127ØØ                MOV     #CØ,RØ
                ØØØ31Ø'
    14 ØØØØ22   Ø127Ø1                MOV     #C1,R1
                ØØØ32Ø'
    15 ØØØØ26   ØØØ167                JMP     INPUT
                ØØØØØØ'
    16 ØØØØØØ                         .PSECT  EING
    17 ØØØØØØ   ØØ4767   INPUT:       JSR     PC,UPEIN
                ØØØØØØG
    18 ØØØØØ4   1Ø3ØØ2                BCC     1¤
    19 ØØØØØ6   ØØØ167                JMP     FEHLER
                ØØØØØØG
    2Ø ØØØØ12            1¤:
    21 ØØØ324                         .PSECT
    22 ØØØ324            PUFFER:      .BLKB   8Ø.
    23 ØØØ444            ZAEHL:       .BLKW   1
    24          ØØØØØØ                .END    ST


PSECT - BEISPIELE          MACRO M28     Ø4-JUL-84 17:25  PAGE 1-1
SYMBOL TABLE

BER      ØØØØØØR          FEHLER= ****** GX         ST       ØØØØØØ       ØØ2
CØ       ØØØ31ØR          INPUT   ØØØØØØR      ØØ3 UPEIN =  ****** GX
C1       ØØØ32ØR          PUFFER  ØØØ324R          ZAEHL    ØØØ444R

. ABS.   ØØØØØØ      ØØØ
         ØØØ446      ØØ1
PA1      ØØØØ32      ØØ2
EING     ØØØØ12      ØØ3
ERRORS DETECTED:  Ø

VIRTUAL MEMORY USED:  1Ø1 WORDS  ( 1 PAGES)
DYNAMIC MEMORY:  3Ø86 WORDS  ( 11 PAGES)
ELAPSED TIME:  ØØ:ØØ:Ø5
,PSECT/LI:TTM=PSECT
```

Bild 1 Beispiel eines Übersetzungsprotokolls mit Einteilung in Programmabschnitte

Tafel 5 Funktionen zur Protokollgestaltung

Argument	Standard	Funktion
SEQ	list	Die Quellzeilennummer wird gedruckt.
LOC	list	Der Speicherplatzzuordnungszähler (SZZ) wird gedruckt.
BIN	list	Der übersetzte Binärkode wird gedruckt.
BEX	list	Der erweiterte Binärkode wird gedruckt. Als erweiterter Binärkode wird der Kode bezeichnet, der in der Zeile der Quellanweisung nicht untergebracht werden kann und somit den Druck von Folgezeilen hervorruft. **.NLIST BIN** verbietet auch **BEX.**
SRC	list	Die Quellzeile wird gedruckt.
COM	list	Die Kommentare, die im Quelltext enthalten sind, werden gedruckt. **.NLIST SRC** verbietet auch **COM.**
MD	list	Makrodefinitionen und die Auflösungen der Wiederholungsblöcke werden gedruckt.
MC	list	Makrorufe und die **.IRP/.IRPC**-Anweisungen werden gedruckt.
ME	no list	Die Makroauflösung wird als Quell- und Binärkode gedruckt.
MEB	no list	Die binäre Makroauflösung wird gedruckt. Das Argument **MEB** ist eine Untermenge des Arguments **ME.**
CND	list	Es wird der Quelltext der nichtgenerierten Bedingungsblöcke und aller Anweisungen der bedingten Generierung gedruckt.
LD	no list	Es werden alle **.LIST**- und **.NLIST**-Anweisungen ohne Argumente gedruckt.
TOC	list	Die Subtiteltabelle wird gedruckt.
TTM	no list	Das Protokollformat wird an das Terminalformat (72 Zeichen pro Zeile) angepaßt. Die Worte der binären Auflösung werden untereinander gedruckt. Bei **.NLIST TTM** wird das Paralleldruckerformat (132 Zeichen pro Zeile) ausgegeben.
SYM	list	Die Symboltabelle wird gedruckt.

titel, den Identifikator des Makroassemblers, Datum und Uhrzeit und eine Seitennummer (Bild 1)
Für die Steuerung des Übersetzungsprotokolls können folgende Anweisungen verwendet werden:

.TITLE – Programmtitel/Modulname
.IDENT – Identifikation des Moduls
.SBTTL – Untertitel des Quellprogramms
.PAGE – Beginn einer neuen Seite
.LIST – Listen
.NLIST – Nicht listen.

Die **.TITLE**-Anweisung dient zur Bezeichnung des Objektmoduls und des Übersetzungsprotokolls mit dem Programmtitel. Sie hat folgende Syntax:

.TITLE *string*

wobei *string* den Programmtitel festlegt. Die ersten 6 Zeichen müssen im Radix-5Ø-Kode zulässig sein und werden als Modulname zur Bezeichnung des Moduls verwendet. Die restlichen Zeichen dürfen beliebige ASCII-Zeichen sein. Die ersten 31 Zeichen des Programmtitels werden in die Kopfzeile eingefügt und auf jedem Blattanfang gedruckt. Wird die **.TITLE-Anweisung** nicht spezifiziert, erhält der Modul den Namen .MAIN.
Die **.IDENT**-Anweisung wird zur zusätzlichen Bezeichnung des Moduls mit einem Versionsidentifikator verwendet. Die Anweisung hat folgende Syntax:

.IDENT */string/*

wobei *string* der Versionsidentifikator aus maximal 6 Radix-5Ø-Zeichen ist, der in Zeichenkettenbegrenzer eingeschlossen werden muß. Als Zeichenkettenbegrenzer können beliebige Sonderzeichen verwendet werden.
Die **.SBTTL**-Anweisung wird zur zusätzlichen Beschriftung des Assemblerprotokolls mit Untertiteln benutzt. Der jeweils letzte Untertitel wird bei Beginn einer neuen Seite als zweite Kopfzeile gedruckt. Vor der ersten Seite des Übersetzungsprotokolls wird eine Zusammenfassung aller Untertitel mit Angabe der Seiten- und Quellzeilennummer als Inhaltsverzeichnis (TOC – Table of contents) gedruckt. Die Anweisung hat folgende Syntax:

.SBTTL *string*

wobei *string* der Untertitel ist. Die Numerierung der Seiten erfolgt durch zwei Nummern: x–y, wobei
x eine logische Seitennummer und
y eine Blattnummer innerhalb der logischen Seite
darstellt.
Die Blattnummer wird ab Null beginnend gezählt, wobei eine Null nicht gedruckt wird. Die logische Seitennummer wird ab Eins beginnend gezählt. Eine Erhöhung der logischen Seitennummer erfolgt durch eine **.PAGE**-Anweisung oder durch Verwendung des Formularsteuerzeichens (FF) als Zeilenbegrenzer.
Die **.PAGE**-Anweisung hat keinen Parameter. Jede neue logische Seite wird auf ein neues Blatt gedruckt, wobei die Blattnummer auf den Anfangswert rückgesetzt wird.
Mit der **.LIST**- bzw. **.NLIST**-Anweisung ohne Argument wird die Protokollierung der Quellzeilen (einschl. Objektkode) erlaubt oder unterdrückt. Die Protokollierung ist erlaubt, wenn ein sogenannter Zähler für das Protokollierungsniveau einen positiven Wert hat. Bei einem negativen Wert erfolgt keine Protokollierung. Der Anfangswert des Zählers ist Null. Durch eine **.LIST**-Anweisung wird der Zähler inkrementiert und durch eine **.NLIST**-Anweisung dekrementiert.
Mit einer **.LIST**- bzw. **.NLIST**-Anweisung mit Argument

.LIST *arg[,arg[,...]]*
.NLIST *arg[,arg[,...]]*

wird die Gestaltung des Protokolls beeinflußt. Der Zähler für das Protokollierungsniveau wird nicht verändert. Die zulässigen Argumente mit ihren Standardwerten sind in Tafel 5 zusammengefaßt.
Die Protokollgestaltung kann auch durch die Schalter **/LI:***arg* und **/NL:***arg* in der Kommandozeile beeinflußt werden. Die durch die Schalter beeinflußten Argumente haben gegenüber den Argumenten der **.LIST**- und **.NLIST**-Anweisung den Vorrang.

2.3. Definitionsanweisungen

Mittels Definitionsanweisungen können Speicherbereiche reserviert oder Zeichenketten und Konstanten in den Objektkode eingetragen werden. Symbole im Namensfeld der Definitionsanweisungen sind erlaubt. Den Symbolen wird der aktuelle Wert des SZZ, d. h. die Anfangsadresse des Bereiches bzw. die Adresse des ersten Bytes der Zeichenkette bzw. der Konstanten, zugeordnet.

2.3.1. Definition von Speicherbereichen

Speicherbereiche können mit den folgenden Anweisungen definiert werden:

.BLKB *expr*
.BLKW *expr*

Die Bereiche werden durch die **.BLKB**-Anweisung in Bytes und durch die **.BLKW**-Anweisung in Worten reserviert, wobei der Ausdruck *expr* die Anzahl der Bytes bzw. Worte festlegt. Der Ausdruck muß absolut sein und darf keine Vorwärtsbezugnahmen oder Bezugnahmen auf externe Symbole enthalten. Die **.BLKW**-Anweisung muß auf einer geradzahligen Adresse beginnen.

Beispiele:
FELD: .BLKW 100
STACK: .BLKB N+OP+2

2.3.3. Definition von Zeichenketten

Die Definition von Zeichenketten im ASCII-Kode bzw. im Radix-50-Kode erfolgt mit den Anweisungen

.ASCII /string/.../string/
.ASCIZ /string/.../string/
.RAD50 /string/.../string/

Mit der **.ASCII**-Anweisung wird ein Speicherbereich reserviert, in den die Kodes der Zeichen der Zeichenkette *string* eingetragen werden, wobei die Zeichen / / die Begrenzer der Zeichenkette darstellen. Die Größe des reservierten Speicherbereiches wird durch die Anzahl der Zeichen festgelegt. Als Begrenzer können beliebige druckbare Zeichen außer <, = und ; benutzt werden. Jedoch darf ein Zeichen, welches innerhalb der Zeichenkette verwendet wird, kein Begrenzer sein. Die Zeichenkette darf in mehrere Teilzeichenketten unterteilt werden. Nichtdruckbare Zeichen dürfen in der Zeichenkette *string* nicht verwendet werden. Sie können durch einen Ausdruck mit dem entsprechenden Wert, der in spitze Klammern eingeschlossen werden muß, definiert werden. Durch einen Ausdruck können beliebige Zeichen definiert werden, jedoch darf der Ausdruck den Wert 377(8) nicht überschreiten. Ausdrücke und Zeichenketten dürfen in beliebiger Reihenfolge definiert werden.

Beispiele:
```
TEXT:.ASCII +BEGINN
        DES PROGRAMMLAUFS+
TEXT1:.ASCII ⟨15⟩⟨12⟩
#LESEFEHLER VON DK0:/DK1 :#
LF=12
CR=15
TEXT2:.ASCII ⟨LF⟩/***FEHLER/
        ⟨CR⟩⟨LF⟩?KODE:?
```

Die **.ASCIZ**-Anweisung entspricht der **.ASCII**-Anweisung, mit dem Unterschied, daß nach dem letzten Byte der Zeichenkette noch ein zusätzliches Byte mit dem Wert 0 als Endekennzeichen definiert wird.
Die **.RAD50**-Anweisung dient der Definition von Zeichenketten im Radix-50-Kode. Auf Grund der Einschränkung des Zeichenvorrates besteht die Möglichkeit, nach der Radix-50-Formel drei Zeichen in zwei Bytes zu definieren. Im Operandenfeld können analog der **.ASCII**-Anweisung Zeichenketten und Ausdrücke in spitzen Klammern verwendet werden. Ein Ausdruck darf den Wert von 47(8) nicht überschreiten. Eine Zeichenkette wird immer in Dreiergruppen unterteilt, die in den Radix-50-Kode umgewandelt und in ein Wort eingetragen werden. Die letzte unvollständige Dreiergruppe wird mit Leerzeichen aufgefüllt.

Beispiele:
A: .RAD50 /ABCDE/
B: .RAD50 ⟨1⟩⟨2⟩⟨3⟩⟨4⟩⟨5⟩
Die Zeichenketten **A** und **B** sind identisch und belegen zwei Worte. Die letzte Dreiergruppe wird durch ein Leerzeichen ergänzt.

2.3.3. Definition von Konstanten

Für alle hardwaremäßig unterstützten Datenformate existiert jeweils eine Definitionsanweisung:

.BYTE *expr[,expr[,...]]*
.WORD *expr[,expr[,...]]*
.FLT2 *arg[,arg[,...]]*
.FLT4 *arg[,arg[,...]]*

Mit der **.BYTE**-Anweisung werden ganze und natürliche Zahlen im Byteformat definiert. Die Anzahl der definierten Bytes entspricht der Anzahl der Ausdrücke. Die Ausdrucksberechnung erfolgt mit 16-Bit-Werten, und anschließend wird das Ergebnis auf 8 Bit verkürzt. Ist der Wert nicht in 8 Bit darstellbar, so wird ein T-Fehler angezeigt. Globale Bezugnahmen sind zulässig.

Beispiele:
```
C1:.BYTE 20,16.,^B10000,2*8.,4+
        ⟨2*6⟩,'0-40
C2: .BYTE ^0377,255,-1
C3: .BYTE W0
    .GLOBL W0
```
Im Bereich **C1** werden 6 Bytes mit dem Wert 20(8) und im Bereich **C2** 3 Bytes mit dem Wert 377(8) definiert. Unter **C3** wird 1 Byte definiert, dessen Wert erst durch den Taskbuilder eingetragen werden kann.

Mit der **.WORD**-Anweisung werden analog der **.BYTE**-Anweisung ganze und natürliche Zahlen im 16-Bit-Format definiert. Eine **.WORD**-Anweisung muß immer auf einer geradzahligen Adresse beginnen. Die Ausdrücke dürfen relative und absolute Werte ergeben sowie globale Bezugnahmen enthalten. Es ist zu beachten, daß alle Anweisungen ohne Operationskode bzw. ohne gültigen Operationskode der **.WORD**-Anweisung gleichgesetzt sind.

Beispiele:
```
V0: .WORD FELD+100., ALPHA
    ;DEFINITION VON 2 ADRESS-
              KONSTANTEN,
.WORD B101110000111, "AB
;DEFINITION VON 2 NUMERISCHEN
              ;KONSTANTEN
```
Diese Anweisungen lassen sich auch wie folgt schreiben:
```
V0: FELD+100., ALPHA
^B101110000111,"AB
```

Unter Anwendung der letzten Regel lassen sich z. B. neue Befehlsmnemoniks für adressenlose Befehle einführen:
```
CLNZ=254; BEFEHL ZUM LOESCHEN
          ;VON N- UND Z-BIT
SVC=104410; TRAP 10
...
...
CLNZ ;GENERIERUNG DER
     ;KONSTANTEN 000254(8)
SVC  ;GENERIERUNG DER
     ;KONSTANTEN 104410(8)
```
Gleitkommazahlen werden mit der **.FLT2**-Anweisung im 2-Wort-Format und mit der **.FLT4**-Anweisung im 4-Wort-Format definiert, wobei jedes Argument zur Generierung einer Gleitkommazahl führt. Die Argumente werden nach den üblichen Regeln mit oder ohne Dezimalpunkt und mit oder ohne Exponent geschrieben. Ausdrücke sind als Argument unzulässig.

Beispiele:
```
FELD: .FLT2 2,+2,2.,2.0,2E0,
        2.0E+0,20.E-1
K0:   .FLT4 .31416E1,
        1.3803E-16,-273.16,
        5.61E26
```
Die erste Anweisung zeigt 7 verschiedene Schreibweisen für die Gleitkommakonstante +2, die noch um zahlreiche Varianten erweitert werden könnte.

2.4. Anweisungen zur Steuerung des Speicherplatzzuordnungszählers

Der Speicherplatzzuordnungszähler (SZZ) wird bei der Assemblierung von Befehlen und Speicherbereichsdefinitionen automatisch um die Länge des Befehls bzw. des Speicherbereiches inkrementiert. Mit der Direktanweisung (siehe 1.7.) kann der SZZ auf einen bestimmten Wert gesetzt bzw. um einen bestimmten Wert weitergezählt werden. Außerdem kann der SZZ durch die

.EVEN-Anweisung und
.ODD-Anweisung

beeinflußt werden.
Die **.EVEN**-Anweisung setzt den SZZ immer auf eine geradzahlige Adresse (Wortgrenze), indem zu einem ungeradzahligen Wert des SZZ eine Eins addiert wird. Ein geradzahliger Wert wird nicht verändert.
Die **.ODD**-Anweisung setzt den SZZ immer auf eine ungeradzahlige Adresse, indem zu einem geradzahligen Wert des SZZ eine Eins addiert wird. Ein ungeradzahliger Wert wird nicht verändert.

Beispiele:
```
TEXT7: .ASCII /EINGABE VON T0:/
       .EVEN
       ;EINSTELLEN DER
       ;WORTGRENZE
DEZ:   .WORD 100.,10.,1
       ;TABELLE IM WORT-
       ;FORMAT
```

```
...
       .ODD ;SZZ UNGERAD-
        ZAHLIG
SATZ:  .BYTE 2*ANZ
       ;SATZLAENGE IM BYTE-
       ;FORMAT
BER:   .BLKW ANZ
       ;DATENSATZ AUF WORT-
       ;GRENZE
```

2.5. *Generelle Assembleranweisungen*

In diesem Abschnitt werden einige Assembleranweisungen von allgemeiner Anwendung, wie

.END – Ende des Quellprogramms
.GLOBL – Definition globaler Symbole
.RADIX – Festlegung der Basis des Zahlensystems

beschrieben.
Die **.END**-Anweisung kennzeichnet das Ende eines Quellmoduls. Sie hat folgende Syntax:

.END *[expr]*

Der Ausdruck *expr* gibt die Startadresse des Programmsystems an. Dadurch wird der automatische Programmstart mit dem RUN-Kommando des Betriebssystems ermöglicht. Wenn mehrere Moduln zu einer Task verbunden werden, so ist die Angabe der Startadresse bei dem Modul notwendig, der als erster gestartet wird. Ist in keinem der Moduln eine Startadresse spezifiziert, so kann der Programmstart nur über das Testhilfesystem (ODT) erfolgen, wenn es in die Task eingebunden wurde.

Die **.GLOBL**-Anweisung wird zur Spezifikation globaler Symbole benutzt, die nicht mittels „: :" bzw. „= =" als globale Symbole definiert wurden. Globale Symbole können globale Definitionen und Bezugnahmen realisieren.

Wird mit **.DSABL GBL** die automatische Definition aller undefinierten Bezugnahmen als globale Referenzen unterdrückt, können globale Referenzen über die **.GLOBL**-Anweisung explizit definiert werden. Die **.GLOBL**-Anweisung hat folgende Syntax:

.GLOBL *symbol[,symbol[,...]]*

wobei
symbol das als global zu spezifizierende Symbol und
, ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen) darstellen.
Die **.RADIX**-Anweisung legt die Basis des Zahlensystems für das Programm fest. Diese Basis wird zugrunde gelegt, wenn im Programm Zahlen ohne Angabe einer temporären Basis (siehe 1.5.) verwendet werden. Die Anweisung hat folgende Syntax:

.RADIX *[n]*

wobei *n* die Basis des Zahlensystems ist, die bis zur nächsten **.RADIX**-Anweisung gilt. Die Dezimalzahl *n* darf die Werte **2, 8** und **10** annehmen. Wenn *n* ausgelassen wird, so wird die Basis **8** angenommen, die ebenfalls gilt, wenn im Programm keine **.RADIX**-Anweisung enthalten ist.

Tafel 6 Argumente der **.PSECT**-*Anweisung*

Argument	Standard	Funktion
RO oder **RW**	**RW**	PA ist nur lesbar (**RO**) oder auch beschreibbar (**RW**).
I oder **D**	**I**	PA mit Befehlen und Daten (**I**) oder PA mit Daten (**D**).
GBL oder **LCL**	**LCL**	Globaler PA einer Überlagerungsstruktur (**GBL**) oder PA mit lokaler Gültigkeit in einem Überlagerungssegment.
ABS oder **REL**	**REL**	Absoluter PA (**ABS**) oder relativer PA (**REL**).
OVR oder **CON**	**CON**	Statische Überlagerung von PAs mit gleichem Namen (**OVR**) oder sequentielle Anordnung von PAs mit gleichem Namen (**CON**).

Tafel 7 Bedingungen der **.IF**/**.IIF**-*Anweisung*

Bedingung positiv	negativ	Argument(e)	Erläuterung
EQ oder Z	**NE** oder NZ	expr	Der Ausdruck wird auf 0 (bzw. ungleich 0) getestet.
GT oder **G**	**LE**	expr	Der Ausdruck wird auf größer als 0 (bzw. kleiner oder gleich 0) getestet.
LT oder **L**	**GE**	expr	Der Ausdruck wird auf kleiner als 0 (bzw. größer oder gleich 0) getestet.
DF	**NDF**	symbol bzw. log. Ausdr.	Es wird getestet, ob das Symbol oder die Symbole entsprechend der logischen Verknüpfung definiert (oder nicht definiert) sind.
B	**NB**	Makroparameter	Es wird geprüft, ob der Makroparameter ausgelassen (blank) oder nicht ausgelassen (no blank) wurde.
IDN	**DIF**	zwei Makroparameter	Es wird geprüft, ob die zwei Makroparameter identisch (oder verschieden) sind.

2.6. *Einleitung von Programmabschnitten*

Ein Programmabschnitt wird durch die Anweisung

.PSECT *[name][,arg[,arg[,...]]]*

eingeleitet, wobei

name – den Namen des Programmabschnittes festgelegt,
, – ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen) und
arg – ein Argument zur Spezifikation von Attributen zur Auswahl von bestimmten Eigenschaften des Programmabschnittes gemäß Tafel 6 bedeuten.

Der Makroassembler läßt in einem Quellmodul 256 Programmabschnitte (PA) zu:

– einen absoluten (nichtverschieblichen) PA, mit dem Namen **. ABS.** (Standard),
– einen relativen (verschieblichen) unbenannten PA (Standard),
– 254 benannte PA.

Für jeden PA wird während der Übersetzung ein SZZ geführt, der ab Null beginnend in relativen oder absoluten Adressen, dem Attribut des PA entsprechend, geführt wird. Somit wird auch den Symbolen im Namensfeld der Anweisungen das Attribut absolut oder relativ zugeordnet. Ein PA muß im Quellprogramm nicht zusammenhängend hintereinander geschrieben werden, sondern kann durch Verwendung der gleichen **.PSECT**-Anweisung weiter unten im Programm fortgesetzt werden. Die Übersetzung wird dann mit dem letzten Stand des SZZ des entsprechenden PA weitergeführt.
Wenn ein Programm ohne **.PSECT**-Anweisung beginnt, so wird ein unbenannter verschieblicher PA aufgebaut, der später auch mit einer **.PSECT**-Anweisung ohne Namen fortgesetzt werden kann.

3. Die bedingte Assemblierung

Die bedingte Assemblierung gestattet die Generierung von verschiedenen Applikationsprogrammen, die an den jeweiligen Anwendungsfall angepaßt sind, aus einem Quellprogramm. Eine grundlegende Bedeutung haben diese sprachlichen Ausdrucksmittel für die Makrotechnik, da sie während der Makrogenerierung die flexible Anpassung der Makros an die verschiedenen Anwendungsfälle gestatten.

3.1. *Die Anweisungen .IF und .ENDC*

Ein bedingter Anweisungsblock wird mit einer **.IF**-Anweisung eingeleitet und mit einer **.ENDC**-Anweisung abgeschlossen. Bedingte Anweisungsblöcke dürfen ineinander geschachtelt werden. Ein bedingter Anweisungsblock hat folgenden Aufbau:

.IF *cond,arg[,arg]*
...
...
Folge von Anweisungen
...
...
.ENDC

wobei die Parameter folgende Bedeutung haben:

cond – spezifiziert entsprechend Tafel 7 eine Bedingung, auf deren Erfüllung das Argument bzw. die Argumente getestet werden. Wenn sich der Wahrheitswert "true" ergibt, so werden die nachfolgenden Anweisungen bis zur **.ENDC**-Anweisung mit übersetzt. Ansonsten werden diese Anweisungen bei der Übersetzung übergangen.
, – zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen)
arg – spezifiziert entsprechend Tafel 7 ein oder zwei Argumente, die Ausdrücke, Symbole oder formale Parameter einer Makrodefinition sein können.

wird fortgesetzt

Programmieren mit MACRO-SM

Teil III

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

In Anweisungen mit den Bedingungen **DF** und **NDF** sind logische Ausdrücke mit den Operatoren **!** und **&** unter Verwendung von spitzen Klammern ⟨ ⟩ zulässig. Die Operatoren haben die gleiche Priorität und werden von links nach rechts abgearbeitet, falls durch eine Klammerung keine andere Abarbeitungsfolge vorgeschrieben wird.

Beispiel:
```
.IF DF M0! ⟨GAMMA&DELTA⟩
...
...
.ENDC
```

Die Anweisungen des bedingten Blockes werden übersetzt, wenn das Symbol **M0** oder die Symbole **GAMMA** und **DELTA** definiert sind.
Die maximale Schachtelungstiefe der bedingten Blöcke beträgt 16. Beim Auftreten einer **.ENDC**-Anweisung außerhalb eines bedingten Blockes bzw. beim Überschreiten der maximalen Schachtelungstiefe wird der Fehlercode 0 angezeigt.

3.2. Die Anweisungen .IFF, .IFT und .IFTF

Die Anweisungen **.IFF**, **.IFT** und **.IFTF** unterteilen einen bedingten Anweisungsblock in Subblöcke und beziehen sich bei geschachtelten bedingten Anweisungsblöcken immer auf die getastete Bedingung der vorangegangenen **.IF**-Anweisung des gleichen Niveaus.
Die **.IFF**-Anweisung bewirkt das Übersetzen der nachfolgenden Anweisungen, wenn die getestete Bedingung den Wahrheitswert "false" ergab.
Die **.IFT**-Anweisung bewirkt das Übersetzen der nachfolgenden Anweisungen, wenn die getestete Bedingung den Wahrheitswert "true" ergab.
Die **.IFTF**-Anweisung bewirkt das Übersetzen der nachfolgenden Anweisungen unabhängig von der getasteten Bedingung. Die einzelnen Subblöcke werden jeweils durch eine nachfolgende **.IFF**-, **.IFT**-, **IFTF**- oder **.ENDC**-Anweisung gleichen Niveaus abgeschlossen.

Beispiel
```
SYMBOL=25
...
.IF    EQ SYMBOL-5
       ;TEST OB SYMBOL-5 GLEICH 0
       IST?
...    ;ANWEISUNGEN WERDEN
       NICHT UEBERSETZT
...
.IFF
...    ;ANWEISUNGEN WERDEN
       UEBERSETZT
...
.IFTF
...    ;ANWEISUNGEN WERDEN
       IMMER UEBERSETZT
...
.IFT
...    ;ANWEISUNGEN WERDEN
       NICHT UEBERSETZT
...
.ENDC
```

3.3. Die .IIF-Anweisung

Die **.IIF**-Anweisung gestattet die bedingte Übersetzung einer einzelnen Anweisung *statement*. Sie hat folgenden Aufbau:

.IIF *cond, arg[,arg], statement*

wobei die Bedingungen, Trennzeichen und Argumente denen der **.IF**-Anweisung entsprechen.
Bei den Bedingungen **B** bzw. **NB** sollte das Argument in spitze Klammern gesetzt werden, da sonst im Falle eines leeren Argumentes (blank) der Parameter statement als *arg* interpretiert wird.

Beispiel:
.IIF DF, ⟨FELD⟩ M0: MOV FELD,R0
Wenn das Symbol **FELD** definiert ist, wird vom Assembler die Anweisung
M0: MOV #FELD,R0
übersetzt. Andernfalls wird die Anweisung nicht übersetzt.

4. Die Makrotechnik

Bei der Programmierung in der Assemblersprache kann der Programmierer oft benötigte Anweisungsfolgen unter Anwendung der Makrotechnik als Makro definieren. Die Makrodefinitionen können Bestandteile des Quellprogramms sein oder in einer Nutzer- oder Systemmakrobibliothek zusammengefaßt werden. Die Bezugnahme auf die Makrodefinitionen erfolgt an den gewünschten Stellen im Programm durch sogenannte Makrorufe, die eine Generierung der in der Makrodefinition enthaltenen Quellanweisungen bewirken. Zur Anpassung des Makros an den jeweiligen Anwendungsfall besteht die Möglichkeit, daß in der Makrodefinition formale Argumente festgelegt werden, die bei der Makrogenerierung durch aktuelle Parameter ersetzt werden können. Die generierten Quellanweisungen werden als Makroauflösung bezeichnet.

4.1. Die Makrodefinition

Eine Makrodefinition beginnt mit einer Makroanfangsanweisung (**.MACRO**), die gleichzeitig die Musteranweisung für den Makroruf ist. Anschließend folgen die Modellanweisungen für die zu generierenden Quellanweisungen. Den Abschluß der Makrodefinition bildet die Makroendanweisung (**.ENDM**). Als Modellanweisungen können in den Makrodefinitionen alle Maschinenbefehle, Assembleranweisungen, Anweisungen der bedingten Übersetzung, Makrorufe und spezielle Anweisungen der Makrotechnik (z. B. **.PRINT**, **.ERROR**, **.MEXIT**, **.NTYPE**, **.NCHR**, **.NARG**) Anwendung finden. Eine Makrodefinition kann im Quellprogramm an beliebiger Stelle stehen, aber es ist zu beachten, daß sie vor ihrem ersten Aufruf bekannt sein muß.

4.1.1. Die .MACRO-Anweisung

Die Makroanfangsanweisung hat folgenden Aufbau:

.MACRO *symbol [,arg [,arg [,. . .]]]*
wobei
symbol – den Namen der Makrodefinition bzw. des Makrorufes festlegt,
, – ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen)
und
arg – ein folgendes Argument bedeuten.

Eine **.MACRO**-Anweisung kann beliebig viele formale Argumente enthalten, die durch wenigstens ein zulässiges Trennzeichen getrennt sein müssen, das heißt, es dürfen ein oder mehrere Tabulatoren oder Leerzeichen, aber nur ein Komma verwendet werden. Als formale Argumente werden Symbole verwendet, auf die sich die Modellanweisungen innerhalb einer Makrodefinition beliebig beziehen können, da sie im Namens-, Operations- oder Operandenfeld oder in Teilen von ihnen oder aber auch als komplette Quellanweisung benutzt werden dürfen. Diese Symbole sind nur interne Symbole einer Makrodefinition.

Beispiel:
.MACRO PUT EABER, LAENGE
Diese Anweisung leitet die Definition des Makros PUT mit den formalen Argumenten **EABER** und **LAENGE** ein.

4.1.2. Die .ENDM-Anweisung

Die Makroendanweisung hat folgenden Aufbau:
.ENDM *[symbol]*
wobei:
symbol – den Namen der zu beendenden Makrodefinition festlegt. Das Symbol hat den Charakter eines Kommentars.

Beispiel:
```
.MACRO   RETT1 P1
MOV      #P1,R5
.MACRO   RETT2
```

```
MOV     R0,(R5)+
MOV     R1,(R5)+
MOV     R2,(R5)+
.ENDM   RETT2 ;ENDE RETT2
MOV     R3,(R5)+
MOV     R4,(R5)+
.ENDM   RETT1 ;ENDE RETT1
```

Bei ineinander geschachtelten Makrodefinitionen kann die innere Makrodefinition erst dann aufgerufen werden, wenn die äußere Makrodefinition wenigstens einmal generiert wurde.

4.1.3. Die .MEXIT-Anweisung

Die **.MEXIT**-Anweisung beendet die Makrogenerierung, bevor das Ende der Makrodefinition erreicht wurde. Besonders vorteilhaft ist die Anwendung der **.MEXIT**-Anweisung in Verbindung mit der bedingten Assemblierung zum Zwecke der bedingten Makrogenerierung bzw. zum Abbruch der Makrogenerierung bei Fehlerbedingungen.

Beispiel:

```
.MACRO  BEISP P1,P2
...
.IF     B,P2
...
...
.MEXIT  ;ENDE MAKROGENERIE-
        ;RUNG, WENN P2 NICHT
        ;SPEZIFIZIERT WURDE
.ENDC
...
.ENDM
```

Wenn das formale Argument **P2** im Makroruf durch keinen aktuellen Parameter ersetzt wird, so werden die Anweisungen nach der **.IF**-Anweisung generiert und danach wird die Generierung abgebrochen. Andernfalls wird der bedingte Block übergangen.

4.1.4. Die .PRINT- und .ERROR-Anweisungen

Die **.PRINT**- und **.ERROR**-Anweisungen werden benutzt, um Mitteilungen in das Assemblerprotokoll auszugeben. Die Anweisungen haben folgendes Format:

.PRINT *[expr] [,text]*

.ERROR *[expr] [,text]*

wobei

expr – zulässiger Ausdruck und

text – die auszugebende Mitteilung bedeuten. Der Wert des Ausdrucks wird berechnet und in der Spalte des Objektkodes ausgedruckt. Dieser Wert kann als eine Mitteilungsnummer betrachtet werden. Durch den Text wird die Mitteilung in der Regel kommentiert. Der Unterschied zwischen den beiden Anweisungen besteht darin, daß bei der **.ERROR**-Anweisung zusätzlich noch das Fehlerkennzeichen P gedruckt wird.

4.2. Der Makroruf

Die Bezugnahme auf eine Makrodefinition erfolgt durch einen Makroruf, der eine Quellzeile mit folgendem Aufbau darstellt:

[symbol:] name liste aktueller parameter

Die Quellzeile kann durch ein Symbol im Namensfeld benannt sein. Im Operationsfeld steht ein Symbol, das den Namen der aufzurufenden und zu generierenden Makrodefinition (siehe **.MACRO**-Anweisung) angibt. Die Liste der aktuellen Parameter im Operandenfeld ist der Reihenfolge der Liste der formalen Argumente entsprechend aufgebaut (Stellungsparameter!), so daß eine entsprechende Substitution formaler Argumente durch aktuelle Parameter möglich ist. Die einzelnen Parameter müssen durch ein zulässiges Trennzeichen (Komma, Leerzeichen und/oder Tabulator) voneinander getrennt werden.

Beispiel:
Aufruf der Makrodefinition aus 4.1.1.:
START: PUT FELD,80.
Durch diesen Makroruf wird der Makro **PUT** aufgerufen, wobei das formale Argument **EABER** durch den aktuellen Parameter **FELD** und **LAENGE** durch **80.** ersetzt werden.

4.3. Die Substitution der Parameter

Die Substitution der formalen Argumente durch aktuelle Parameter erfolgt in der Reihenfolge ihrer Stellung (Stellungsparameter). Werden im Makroruf mehr Parameter als notwendig angegeben, so werden die nicht benötigten Parameter nicht berücksichtigt. Werden im Makroruf zu wenige Parameter angegeben oder Parameter ausgelassen, so werden die entsprechenden formalen Argumente als leer (*blank*) angenommen und bei der Makrogenerierung durch nichts ersetzt. Enthält ein aktueller Parameter Trennzeichen, so muß er in spitze Klammern eingeschlossen werden.

Beispiel:
BEISP FELD, ⟨MOV A,–(SP)⟩
Das erste formale Argument soll durch die Adresse **FELD** und das zweite durch den Befehl **MOV A,–(SP)** ersetzt werden. Da der zweite Parameter Trennzeichen (Leerzeichen und Komma) enthält, muß er in spitze Klammern eingeschlossen werden.

4.3.1. Die Parameterübergabe bei ineinander geschachtelten Makrorufen

In Makrodefinitionen können Makrorufe benutzt werden, wobei eine maximale Schachtelungstiefe von 16 nicht überschritten werden darf. In einem Makroruf, der sich innerhalb einer Makrodefinition befindet, können als aktuelle Parameter die formalen Argumente der Makrodefinition benutzt werden.

Beispiel:

```
.MACRO  MAC1 P1,P2,P3
...
...
MAC2    P2
...
.ENDM
```

Die Makrodefinition **MAC2** kann folgendermaßen definiert sein:

```
.MACRO  MAC2 P1,P2
...
...
.ENDM
```

Wenn der Makro **MAC1** wie folgt aufgerufen wird,
MAC1 #100,ALPHA, ⟨MOV R0,X0⟩
so wird der Makroruf
MAC2 ALPHA
generiert.

Unter Anwendung von spitzen Klammern können an den Makroruf **MAC2** auch mehrere Parameter übergeben werden.

Beispiel:
MAC1 #100, ⟨ALPHA,BETA⟩,
⟨MOV R0,X0⟩
Es wird der Makroruf
MAC2 ALPHA,BETA
generiert.

Die äußeren spitzen Klammern werden bei der Parameterübergabe nicht mit übergeben. Wenn die Notwendigkeit besteht, daß die spitzen Klammern mitgeneriert werden, so kann eine Klammerung mit der sogenannten Pfeilkonstruktion ^*x* ... *x* erfolgen. Durch den einstelligen Pfeiloperator wird das nachfolgende Zeichen *x* zum Trennzeichen zur Klammerung des Ausdruckes erklärt. Es kann ein beliebiges Zeichen sein, das im Klammerausdruck selbst nicht benutzt wird.

Beispiel:
Bei dem Makroruf
MAC1 #100,^&⟨MOV2(R5),C+6⟩,
F0&,⟨MOV R0,X0⟩
wird der Makroruf
MAC2 ⟨MOV 2(R5),C+6⟩ ,F0
generiert.

Es besteht auch andererseits die Möglichkeit, das formale Argument der Makrodefinition, das als aktueller Parameter eines Makrorufes verwendet wird, in spitze Klammern oder in eine Pfeilkonstruktion einzuschließen, so daß die Übergabe der Klammern nicht unbedingt mit der Parameterübergabe organisiert werden muß.

Beispiel:

```
.MACRO  MAC1 P1,P2,P3
...
...
MAC2    ⟨P2⟩
...
.ENDM
```

Bei dem Makroruf
MAC1 #100,⟨MOV 2 (R5),C+6⟩,⟨MOV R0,X0⟩
wird der Makroruf
MAC2 ⟨MOV 2(R5),C+6⟩
generiert.

4.3.2. Die Verkettung der Parameter

In Makrodefinitionen ist der Apostroph **(')** als Trennzeichen zur Begrenzung der formalen Argumente zugelassen. Bei der Makrogene-

rierung wird der Apostroph als **Begrenzer** erkannt und entfernt, sofern er unmittelbar vor oder hinter einem formalen Argument steht. Diese Eigenschaft des Apostrophs wird genutzt, um aktuelle Parameter untereinander oder mit anderen Zeichen zu verketten. Wenn ein Apostroph erhalten bleiben soll, so sind in der Makrodefinition zwei Apostrophe zu schreiben.

Beispiel:
```
.MACRO MAC3 P1,P2,P3
P1'P3: .RAD50 /P1'P2'P3'0/
       .BYTE "P1,"P2
       .ENDM
```
Wenn der Makro durch folgende Quellanweisung aufgerufen wird:
MAC3 A,B,1Z2
so werden die Anweisungen
```
A1Z2: .RAD50 /AB1Z20/
      .BYTE 'A,'B
```
generiert.

4.3.3. Numerische Symbole

Zur Bildung von Symbolen in Makrodefinitionen können auch numerische Symbole benutzt werden. Dadurch wird es möglich, daß z. B. beim mehrmaligen Aufruf eines Makros doppelt definierte Symbole vermieden werden. Ein numerisches Symbol wird von einem aktuellen Parameter gebildet, wenn dem aktuellen Parameter der Operator \ (Backslash) im Makroruf vorangestellt wird. Der Operator \ bewirkt, daß der numerische Wert des aktuellen Parameters zum Zeitpunkt des Aufrufes des Makros in eine Zeichenkette entsprechend der aktuellen Basis des Zahlensystems umgewandelt wird, wobei Vornullen unterdrückt werden.

Beispiel:
```
.MACRO DEFW P1,P2
DEFWD  P1,P2, \MACIND
MACIND=MACIND+1

.ENDM
.MACRO DEFWD Z1,Z2,Z3
M'Z3:   .WORD Z1,Z2
        .ENDM
```
Bevor der Makro **DEFW** das erste Mal aufgerufen wird, muß **MACIND** definiert werden:
MACIND=0.
Beim ersten Aufruf des Makros
DEFW FELD,50.
werden die folgenden Anweisungen generiert:
```
m0:     .WORD FELD,50.
MACIND=MACIND+1
```
Es wird als Symbol **m0** generiert. Anschließend wird **MACIND** um 1 erhöht. Beim nächsten Aufruf des Makros **DEFW** wird dann das Symbol **M1** gebildet usw.

4.3.4. Die automatische Generierung lokaler Symbole

In Makrodefinitionen werden oft interne Symbole benötigt. Dafür können lokale Symbole benutzt werden. Der Makroassembler ersetzt formale Argumente automatisch durch lokale Symbole, wenn das formale Argument in der **.MACRO**-Anweisung durch ein vorangestelltes Fragezeichen (**?**) gekennzeichnet und im Makroruf das formale Argument durch keinen aktuellen Parameter ersetzt wird. Für die Makrogenerierung sind die lokalen Symbole **64**○ bis **127**○ reserviert. Es wird automatisch immer das nächste freie Symbol belegt. Wenn die lokalen Symbole erschöpft sind, so kann vom Programmierer durch eine **.ENABL-LSB**-Anweisung ein neuer lokaler Symbolblock eröffnet werden.

Beispiel:
```
.MACRO MAC4 P1,P2,?P3,?P4
MOV        P1,R1
P3:        CMP R0,(R1)+
           BMI P4
           DEC P2
           BNE P3
P4:
.ENDM
```
Durch den Makroruf
MAC4 FELD,ZAHL,,MARKE
werden folgende Anweisungen generiert:
```
MOV       #FELD,R1
64○:      CMP R0,(R1)+
          BMI MARKE
          DEC ZAHL
          BME 64○
MARKE:
```

4.4. Die Makroattributanweisungen

In der Makroassemblersprache MACRO-SM sind die Makroattributanweisungen **.NARG**, **.NCHR** und **.NTYPE** realisiert. Die Anweisung **.NARG** wird in Makrodefinitionen benutzt, um die Anzahl der im Makroruf spezifizierten aktuellen Parameter zu bestimmen. Die Anweisung hat folgende Syntax:

.NARG *symbol*

wobei dem Symbol ein Wert zugewiesen wird, der der Anzahl der aktuellen Parameter des Makrorufes entspricht.
Die Anweisung **.NCHR** kann in Assemblerprogrammen und in Makrodefinitionen benutzt werden, um die Anzahl der Zeichen einer Zeichenkette zu bestimmen. Besondere Bedeutung hat die Anweisung für die Bestimmung der Zeichenkettenlänge von aktuellen Parametern in Makrorufen. Sie hat folgende Syntax:

.NCHR *symbol[,⟨string⟩]*

wobei:
symbol– ein Symbol ist, dem ein Wert zugewiesen wird, der der Anzahl der Zeichen der Zeichenkette ⟨*string*⟩ entspricht;
, – ein zulässiges Trennzeichen (Komma, Leerzeichen und/oder Tabulator) ist;
⟨*string*⟩ – die Zeichenkette spezifiziert, von der die Länge bestimmt werden soll. Enthält die Zeichenkette Trennzeichen, so muß sie in spitze Klammern oder in eine Pfeilkonstruktion ˆ*x* . . . *x* eingeschlossen werden. Die spitzen Klammern bzw. die Begrenzer der Pfeilkonstruktion werden nicht mitgezählt. Wenn die Zeichenkette spitze Klammern enthält, so muß die Zeichenkette mit einer Pfeilkonstruktion eingeschlossen werden. Wird in der Anweisung keine Zeichenkette spezifiziert, so wird dem Symbol der Wert 0 zugewiesen.

Beispiel:
```
.NCHR  ZZ,<ALPHA=>
             ;DEM SYMBOL ZZ WIRD
             ;DER WERT 6 ZUGEWIE-
             ;SEN
NCHR   X0,^/PC UND SP <CR>/
             ;DEM SYMBOL ZZ WIRD
             ;DER WERT 15 ZUGEWIE-
             ;SEN
```

Die Anweisung **.NTYPE** wird in Makrodefinitionen benutzt, um den Adressierungsmodus eines aktuellen Parameters eines Makrorufes zu bestimmen. Sie hat folgenden Aufbau:

.NTYPE *symbol[,expr]*

wobei:
symbol – ein Symbol ist, dem ein Wert zugewiesen wird, der dem Adressierungsmodus des Adreßausdrucks entsprechend Tafel 8 charakterisiert ist (6 Bit);
, – ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen) ist;
expr – ein zulässiger Adreßausdruck ist, wie er auch in symbolischen Maschinenbefehlen benutzt werden kann. Wenn kein Ausdruck spezifiziert wird, so wird dem Symbol der Wert 0 zugewiesen.

Beispiel:
```
.MACRO  LAD P1
.NTYPE  MOD,P1
.IF     EQ,MOD-27
MOV     P1,R4
.MEXIT
.ENDC
.IF     NE,MOD-67
.ERROR  ;FALSCHER ADRESSIE-
        ;RUNGSMODUS
.MEXIT
.ENDC
MOV     #P1,R4
.ENDM
```
Bei dem Makroruf
LAD #FELD+2
wird der Befehl MOV #FELD+2,R4 generiert.
Bei dem Makroruf
LAD FELD+2
wird der gleiche Befehl wie beim vorhergehenden Aufruf generiert. Bei allen anderen Adressierungsmodi wird kein Befehl generiert und die Generierung mit einem P-Fehler abgebrochen.

4.5. Makrodefinitionen aus der Makrobibliothek

Für den Anwender besteht die Möglichkeit, Makrodefinitionen mit dem Bibliothekar LBR in einer Makrobibliothek abzuspeichern. Außerdem kann der Nutzer auf die Systemmakrobibliothek zurückgreifen. Der Aufruf der Makrodefinitionen aus den Makrobibliotheken erfolgt mit der **.MCALL**-Anweisung vor der ersten Generierung. Sie hat folgende Syntax:

.MCALL *arg [, arg, [, . . .]]*

Tafel 8 Übersicht über die Adressierungsarten

Notation	Numerischer Wert	Bezeichnung	Bedeutung
R	0n	Registeradressierung	Das Register R enthält den Operanden.
(ER) oder @R	1n	Register-Indirekt-Adressierung	Das Register R enthält die Adresse des Operanden.
(ER)+	2n	Autoinkrement-Adressierung	Das Register, spezifiziert durch ER, enthält die Adresse des Operanden. Nach der Operation wird der Registerinhalt erhöht.
@(ER)+	3n	Autoinkrement-Indirekt-Adressierung	Das Register, spezifiziert durch ER, enthält die Adresse der Adresse des Operanden. Nach der Operation wird der Registerinhalt erhöht.
−(ER)	4n	Autodekrement-Adressierung	Das Register, spezifiziert durch ER, enthält die Adresse des Operanden, die vor der Operation erniedrigt wird.
@−(ER)	5n	Autodekrement-Indirekt-Adressierung	Das Register, spezifiziert durch ER, enthält die Adresse der Adresse des Operanden. Vor der Operation wird der Registerinhalt erniedrigt.
E(ER)	6n	Index-Adressierung	Der Ausdruck E plus der Inhalt des Registers, spezifiziert durch ER, ergeben die Adresse des Operanden. Die Indexkonstante E steht nach dem OPC im Speicher.
@E(ER)	7n	Index-Indirekt-Adressierung	Der Ausdruck E plus der Inhalt des Registers, spezifiziert durch ER, ergeben die Adresse der Adresse des Operanden. Die Indexkonstante E steht nach dem OPC im Speicher.
#E	27	Direktwertadressierung	Der Ausdruck E ist der Operand selbst. Der Direktwert steht nach dem OPC im Speicher.
@#E	37	Absolute Adressierung	Der Ausdruck E ist die Adresse des Operanden. Nach dem OPC steht die absolute Adresse im Speicher.
E	67	Relative Adressierung	Der Ausdruck E ist die Adresse des Operanden. Nach dem OPC steht die relative Adresse als Differenz der Operandenadresse und des Befehlszählers im Speicher.
@E	77	Indirekte relative Adressierung	Der Ausdruck E ist die Adresse der Adresse des Operanden. Nach dem OPC steht die relative Adresse als Differenz der Adresse der Adresskonstante und des Befehlszählers im Speicher

n – Natürliche Zahl zwischen 0 und 7 als Registernummer
R – Registerausdruck (Register expression)
E – Ausdruck (Expression)
ER – Registerausdruck oder Ausdruck, der einen Wert von 0 bis 7 ergibt.

wobei *arg* für die Namen der benötigten Makrodefinitionen steht. Die Suche der Makrodefinitionen erfolgt zunächst in den Nutzerbibliotheken, die in der Kommandozeile für den Makroassembler spezifiziert wurden. Danach wird die Suche in der Systemmakrobibliothek fortgesetzt. Wird eine Makrodefinition nicht gefunden, so wird das Fehlerzeichen U gesetzt.

4.6. Indirekte Makrodefinitionen

Unter einer indirekten Makrodefinition wird die Definition eines Wiederholungsblockes verstanden, die eine zyklische Generierung der Anweisungen des Wiederholungsblockes an der Stelle im Programm bewirkt, wo der Wiederholungsblock definiert ist. Wiederholungsblöcke können im Assemblerprogramm und in Makrodefinitionen angewendet werden.
Wiederholungsblöcke werden durch eine **.IRP**-, **.IRPC**- oder **.REPT**-Anweisung eingeleitet und mit einer **.ENDM**-Anweisung abgeschlossen. Für die Substitution formaler Argumente durch aktuelle Parameter gelten die gleichen Regeln wie für Makrodefinitionen (siehe 4.3.).

4.6.1. Die .IRP-Anweisung

Bei der **.IRP**-Anweisung wird ein formales Argument nacheinander durch mehrere aktuelle Parameter ersetzt. Dabei werden die Modellanweisungen des Wiederholungsblockes für jeden aktuellen Parameter einmal generiert. Die Anzahl der Wiederholungen wird somit durch die Anzahl der aktuellen Parameter bestimmt. Der Wiederholungsblock mit der **.IRP**-Anweisung hat folgenden Aufbau:

.IRP *arg*, ⟨*par1*,*par2*,...⟩
...
...
Modellanweisungen
...
...
.ENDM
wobei
arg – ein Symbol darstellt, das als formales Argument benutzt wird;
, – ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen) ist;
⟨*par1*,*par2*, ...⟩ – die Liste der aktuellen Parameter darstellt. Die aktuellen Parameter müssen durch ein zulässiges Trennzeichen getrennt und in spitze Klammern eingeschlossen werden. Die einzelnen Parameter können bei Notwendigkeit selbst in spitze Klammern oder in eine Pfeilkonstruktion eingeschlossen werden.

Beispiel:
```
.IRP  P1, ⟨ALPHA,BETA,GAMMA⟩
 ADD  P1,R0
.ENDM
```
Es werden folgende Anweisungen generiert:
```
 ADD  ALPHA,R0
 ADD  BETA,R0
 ADD  GAMMA,R0
```

4.6.2. Die .IRPC-Anweisung

Bei der **.IRPC**-Anweisung wird ein formales Argument nacheinander durch das jeweils nächste Zeichen einer Zeichenkette ersetzt. Die Anzahl der Generierungen des Wiederholungsblockes wird durch die Länge der Zeichenkette bestimmt. Der Wiederholungsblock mit der **.IRPC**-Anweisung hat folgenden Aufbau:

.IRPC *arg*,⟨*string*⟩
...
...
Modellanweisungen
...
...
.ENDM
wobei
arg – ein Symbol darstellt, das als formales Argument benutzt wird;
, – ein zulässiges Trennzeichen (Komma, Tabulator und/oder Leerzeichen) ist;
⟨*string*⟩ – die Zeichenkette darstellt, deren Zeichen als aktuelle Parameter eingesetzt werden. Die Zeichenkette muß in spitze Klammern eingeschlossen werden, wenn sie Trennzeichen enthält.

Beispiel:
```
.IRPC  P0,ERG=
 MOVB#"P0,(R2)+
.ENDM
```
Es werden folgende Anweisungen generiert:
```
MOVB  #'E,(R2)+
MOVB  #'R,(R2)+
MOVB  #'G,(R2)+
MOVB  #'=,(R2)+
```

4.6.3. Die .REPT-Anweisung

Die **.REPT**-Anweisung wird benutzt, um eine Anzahl von Quellanweisungen mehrfach unverändert zu generieren. Der Wiederholungsblock mit der **.REPT**-Anweisung hat folgenden Aufbau:

.REPT *expr*
...
Quellanweisungen
...
.ENDM (auch **.ENDR**)
wobei
expr – ein zulässiger Ausdruck ist, der die Anzahl der Wiederholungen angibt. Wenn der Ausdruck den Wert 0 ergibt, so wird keine Anweisung generiert.

(wird fortgesetzt)

Programmieren mit MACRO–SM

Teil IV
Dr. Thomas Horn
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

5. Der Befehlssatz

In diesem Abschnitt werden die wesentlichen Kenntnisse über den Befehlssatz der Rechenanlagen des SKR für den Programmierer in MACRO-SM zusammengefaßt. Das Format der symbolischen Maschinenbefehle entspricht dem gemäß Punkt 1.1. dargelegten allgemeinen Format von Assembleranweisungen. Als Voraussetzung für die Anwendung der Maschinenbefehle werden zunächst die Registerstruktur, die Befehlsformate und die Adressierungsarten betrachtet.

5.1. Die Registerstruktur der Prozessoren

Dem Programmierer steht ein Registerblock mit 8 allgemeinen Mehrzweckregistern und das Prozessorstatusregister zur Verfügung. Die 8 allgemeinen Register können direkt durch die Befehle adressiert werden. Sie werden mit einer 3-Bit-Nummer kodiert. Man nennt sie deshalb gewöhnlich Register 0 (**R0**), Register 1 (**R1**), ..., Register 7 (**R7**). Diese Register können als Akkumulatoren, Adressierungsregister, Indexregister, Kellerzeiger (Stackpointer) und Zwischenregister zur Speicherung von Daten und Adressen benutzt werden.
Die Register haben eine Länge von 16 Bit (1 Wort). Von den Byteverarbeitungsbefehlen werden die niederwertigen 8 Bit der Register beeinflußt. Eine Ausnahme bildet nur das Laden eines Bytes, wobei die höherwertigen 8 Bits des Registers auf 0 oder 1 gesetzt werden, entsprechend dem Vorzeichen des zu ladenden Bytes. Dadurch erfolgt die automatische Umwandlung von Daten aus dem Byte- in das Wortformat.
Die Register R6 und R7 nehmen unter den allgemeinen Registern eine Sonderstellung ein. Das Register R7 wird von der Gerätetechnik als Befehlszähler (**PC** – Program counter) benutzt. Der **PC** enthält immer die Adresse des nächsten abzuarbeitenden Befehls. Daraus ergeben sich einige Einschränkungen für die Benutzung des Registers R7, da nur einige Operationen sinnvoll sind. Zum Beispiel ist das Laden der Konstante 1000 in das Register R7 identisch mit einem Sprung zur Adresse 1000; dagegen ist das Bilden des Komplements vom Inhalt des Registers R7 eine absurde Operation, die zu einer Programmausnahme führt.
Das Register R6 wird als Stackpointer (**SP**) für einen sogenannten Systemstack benutzt. Dieser Systemstack ist für die Unterprogramm- und Interruptorganisation notwendig. Der Systemstack wird vom Betriebssystem durch Laden der Adresse eines speziell dafür reservierten Hauptspeicherbereiches in den **SP** eingerichtet. Bei einem Unterprogrammruf wird in den Systemstack die Rücksprungadresse zur späteren Fortsetzung des Hauptprogramms eingetragen. Bei einem Interrupt wird der Inhalt des **PC** und des Prozessorstatusregisters in den Systemstack gerettet, so daß nach der Interruptbehandlung das unterbrochene Programm fortgesetzt werden kann. Da der Systemstack einen allgemeinen temporären Arbeitsspeicherbereich darstellt, wird von ihm außerdem bei der Programmierung zum Retten von Registerinhalten, zur Parameterübermittlung bei der Unterprogrammorganisation usw. Gebrauch gemacht.
Das Prozessorstatusregister (**PS**) ist ein 16-Bit-Register, das die Bedingungsflags und die Priorität des laufenden Programms beinhaltet. Auf das **PS** kann nur über seine Busadresse 777776(8) zugegriffen werden. Eine Ausnahme sind der K 1620 und die Elektronika-60, bei denen der Zugriff auf das **PS** über 2 spezielle Befehle (**MTPS, MFPS**) erfolgt. Die Bedingungsflags werden von der arithmetisch-logischen Einheit gesetzt und von den bedingten Verzweigungsbefehlen ausgewertet. Die Programmpriorität wird vom Programm oder Betriebssystem gesetzt und von der Interruptsteuerung benutzt.

5.2. Der Befehlsaufbau

Ein Befehl belegt in seiner Grundstruktur 16 Bit. Bei verschiedenen Adressierungsmodifikationen wird der Befehl durch Direktwerte, relative und/oder absolute Adressen auf zwei oder drei Worte erweitert. Die Befehle können in verschiedene Befehlsgruppen unterteilt werden:
1. Ein-Adreß-Befehle
2. Zwei-Adreß-Befehle
3. Verzweigungsbefehle
4. Spezialbefehle (adressenlose Befehle).

Die Befehle bestehen im allgemeinen aus einem Operations- und einem Operandenteil. Da ein Befehl in der Grundstruktur nur 16 Bit belegt, ist die Angabe von echten Hauptspeicheradressen nicht möglich. Deshalb erfolgt die Operandenadressierung bei den Ein- und Zwei-Adreß-Befehlen über die allgemeinen Register. Zusätzlich zur Registerangabe wird bei den Befehlsstrukturen ein Adressierungsmodus spezifiziert, der die Art und Weise der Verwendung des Registers angibt. Die sich daraus ergebenden verschiedenen Adressierungsarten werden nachfolgend detailliert betrachtet.
Die meisten Ein- und Zwei-Adreß-Befehle sind für die Verarbeitung von Operanden im Wort- und Byteformat realisiert (Wort- und Byteverarbeitungsbefehle). Zur Unterscheidung dieser beiden Befehlstypen wird das Bit 15 des Befehlswortes benutzt:
Bit 15 = 0 – Wortverarbeitungsbefehl
Bit 15 = 1 – Byteverarbeitungsbefehl.

5.2.1. Ein-Adreß-Befehle

Der Aufbau der Ein-Adreß-Befehle ist in Bild 2 dargestellt. Die meisten Ein-Adreß-Befehle sind vom Typ 1 und enthalten einen Operationskode (10 Bit) und die Beschreibung der Zieladresse, bestehend aus dem Modus (3 Bit) und der Registernummer (3 Bit).
Dagegen sind nur wenige Ein-Adreß-Befehle vom Typ 2. Dazu zählt der Befehl „Rücksprung aus dem Unterprogramm".
Bei diesen Befehlen wird der Modus der Verwendung des Registers durch den Operationskode festgelegt.
Beispiel:
Löschen des Registers 3
Bei diesem Befehl steht der Operand direkt im allgemeinen Register. Die direkte Registeradressierung ist der Modus 0. In dualer Schreibweise:

0 000 101 000	000	011
Operationskode	Modus	Register

In oktaler Schreibweise: **005003**
In symbolischer Schreibweise: **CLR R3**

5.2.2. Zwei-Adreß-Befehle

Der Aufbau der Zwei-Adreß-Befehle ist in Bild 3 dargestellt. Die meisten Zwei-Adreß-Befehle sind vom Typ 1. Der Operationskode umfaßt 4 Bit und die beiden Adreßangaben, Quell- und Zieladresse, jeweils 6 Bit. Sie bestehen aus einer Register- und Modusangabe.
Es gibt nur wenige Zwei-Adreß-Befehle vom

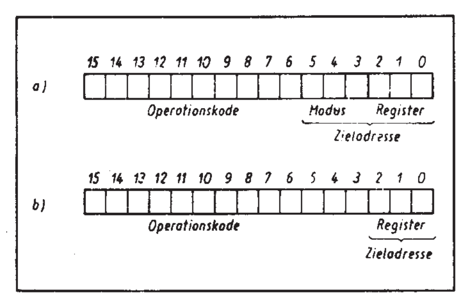
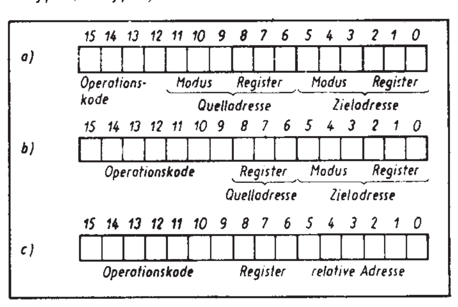


Bild 2 Aufbau der Ein-Adreß-Befehle (a-Typ 1, b-Typ 2)

Bild 3 Aufbau der Zwei-Adreß-Befehle (a-Typ 1, b-Typ 2, c-Typ 3)



Typ 2. Dazu zählen der „Unterprogrammsprungbefehl“ und die Befehle des erweiterten Befehlssatzes. Das Register für die Quelloperandenangabe wird dem Modus 0 entsprechend verwendet (Operand im Register). Es gibt nur einen Zwei-Adreß-Befehl vom Typ 3, den Zyklus-Befehl. Das Register wird zur Zählung der Schleifen benutzt, und die relative Adresse stellt eine Sprungdistanz (in Worten!) zum Schleifenanfang dar. Die relative Adresse wird somit doppelt vom aktuellen Befehlszählerstand subtrahiert. Es ist zu beachten, daß dabei der Befehlszähler (PC) schon auf den nächsten Befehl zeigt.

Beispiel:
Transport eines Operanden aus Register 5 in das Register 3. Der Adressierungsmodus ist auf Grund der direkten Registeradressierung für beide Adressen 0.
In dualer Schreibweise:

0 001	000	101	000	011
Operationskode	Modus	Reg.	Modus	Reg.
	Quelladr.		Zieladresse	

In oktaler Schreibweise: **0 1 0 5 0 3**
In symbolischer Schreibweise: **MOV R5, R3**

5.2.3. Verzweigungsbefehle

Der Aufbau der Verzweigungsbefehle ist in Bild 4 dargestellt: Im höherwertigen Byte steht der Operationskode und im niederwertigen Byte eine Verschiebung (relative Adresse), die eine Sprungdistanz (in Worten!) im Zweierkomplement zum Sprungziel darstellt. Addiert man die Verschiebung doppelt zum aktuellen Befehlszählerstand, so erhält man die Adresse des Sprungzieles. Es ist darauf zu achten, daß der Befehlszähler dabei schon auf den nächsten Befehl zeigt.

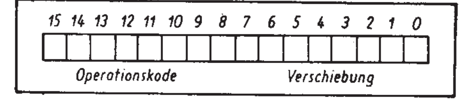


Bild 4 Aufbau der Verzweigungsbefehle

5.2.4. Spezialbefehle

Unter diese Kategorie fallen alle adressenlosen Befehle, wie:
- Steuerbefehle (**HALT, WAIT** usw.)
- Setzen/Löschen der Bedingungsflags (SEC, CLC, ...)
- Trap-Befehle (Unterbrechungsbefehle).

Bei den Bedingungsflag- und Steuerbefehlen belegt der Operationskode alle 16 Bit des Befehlswortes. Beim **TRAP-** und **EMT-**Befehl enthält das niederwertige Byte die Nummer des **TRAP-** oder **EMT-**Befehls.

5.3. Adressierungsarten

Die Adressierungsart wird durch den Adressierungsmodus bestimmt, der im Befehl in der Regel für jeden Operanden mit 3 Bit kodiert wird. Daraus ergeben sich die nachfolgend beschriebenen 8 Grundadressierungsarten, die gerätetechnisch realisiert sind. Als erweiterte Adressierungsart werden die Adressierungsarten bezeichnet, die sich programmtechnisch unter Anwendung des Befehlszählers (**PC**) als Adressierungsregister und einiger Grundadressierungsarten organisieren lassen (auch **PC**-Adressierungsmodifikationen genannt). Sie werden programmtechnisch durch den Assembler unterstützt, der bei der Übersetzung der speziellen Notationen den entsprechenden Maschinenkode erzeugt, der sich auf die beschriebenen Grundadressierungsmodifikationen zurückführen läßt. In den Assemblernotationen werden folgende Kurzbezeichnungen verwendet:

E – ist ein Ausdruck gemäß 3.6, der eine 16-Bit-Adresse ergibt, die entsprechend dem Charakter des PA absolut bzw. absolut oder relativ sein kann, z. B. **ALPHA+2.**

R – ist ein Registerausdruck, der entweder das Registerkennzeichen % oder ein Registersymbol enthält, z. B. **PC, R0, %REG+3, %10/2, R2+1, %2+1.**

ER – ist ein Registerausdruck oder Ausdruck, der einen Wert von 0 bis 7 ergibt.

Die Verwendung eines Ausdrucks zur Bezeichnung eines Registers ist nur dann eindeutig, wenn der Ausdruck in runde Klammern eingeschlossen ist, die der Bedeutung „Register-Indirekt“ entsprechen.
Eine zusammenfassende Übersicht über die Adressierungsarten ist in Tafel 9 enthalten.

5.3.1. Die Grundadressierungsarten

Registeradressierung (Modus 0)
Bei der Registeradressierung (Bild 5) wird der Registerinhalt direkt als Operand benutzt. Da die allgemeinen Register integrierter Bestandteil des Prozessors sind, ergibt diese Adressierungsart die kürzeste Befehlsausführungszeit. Für den Zugriff zum Operanden wird kein Buszyklus benötigt.
Das Register *R* enthält den Quell- oder Zieloperanden.

Beispiele:
```
INC R2; ERHOEHEN DES
;REGISTERINHALTES UM 1
CLR %2; LOESCHEN DES REGISTERS %2
```

Register-Indirekt-Adressierung (Modus 1)

Bei der Register-Indirekt-Adressierung (Bild 6) wird der Inhalt des adressierten Registers als die Adresse des Operanden interpretiert. Für den Zugriff zum Operanden wird ein Buszyklus benötigt.
Das Register enthält die Adresse des Operanden im Hauptspeicher. Der Assembler läßt auch die Notation @*R* zu, wobei aber nur ein Registerausdruck *R* zulässig ist.

Beispiele:
```
INC (2) ; ADRESSE IN REGISTER 2
CLR @%2
CLR @R2
CLR (R2)
```

Autoinkrement-Adressierung (Modus 2)

Bei der Autoinkrement-Adressierung (Bild 7) wird wie bei der Register-Indirekt-Adressierung der Inhalt des adressierten Registers als die Adresse des Operanden interpretiert.
Für den Zugriff zum Operanden wird ein Buszyklus benötigt. Anschließend wird der Registerinhalt um die Länge des Operanden erhöht, bei Byteverarbeitungsbefehlen um 1 und bei Wortverarbeitungsbefehlen um 2.
Bei den Registern 6 und 7 erfolgt immer eine Erhöhung um 2.
Das Register *ER* enthält die Adresse des

Datenformate des SKR

1. Festkommazahlen im Byteformat
a) Natürliche Zahlen
7 … 0
Mantisse
Wertebereich: 0 bis 255.
b) Ganze Zahlen
7 … 0
V Mantisse
Vorzeichen (V): 0 – positiv, 1 – negativ
Wertebereich: – 128. bis + 127.

2. Festkommazahlen im Wortformat
a) Natürliche Zahlen
15 … 0
Mantisse
Wertebereich: 0 bis 65535.
b) Ganze Zahlen (I-Format)
15 … 0
V Mantisse
Vorzeichen (V): 0 – positiv, 1 – negativ
Wertebereich: – 32768. bis + 32767

3. Ganze Zahlen im Langwortformat (L-Format)
15 … 0
V Mantisse
15 … 0
Mantisse
Vorzeichen (V): 0 – positiv, 1 – negativ
Wertebereich: – 2147483648. bis + 2147483647.

4. Gleitkommazahlen einfacher Genauigkeit (F-Format)
15 14 … 7 6 … 0
V Charakteristik Mantisse (höherwertiger Teil)
15 … 0
Mantisse (niederwertiger Teil)
Vorzeichen (V): 0 – positiv, 1 – negativ
Charakteristik = 200 (8) + Exponent
Das höherwertige Bit der Mantisse wird nicht dargestellt.
Wertebereich: 0.2938736 * 10–38 bis 0.1701412 * 10+39
Genauigkeit: 24 Bit (7 Dezimalziffern)

5. Gleitkommazahlen doppelter Genauigkeit (D-Format)
15 14 … 7 6 … 0
V Charakteristik Mantisse (1. Teil)
15 … 0
Mantisse (2. Teil)
15 … 0
Mantisse (3. Teil)
15 … 0
Mantisse (4. Teil)
Vorzeichen (V): 0 – positiv, 1 – negativ
Charakteristik = 200 (8) + Exponent
Das höherwertige Bit der Mantisse wird nicht dargestellt.
Wertebereich: 0.29387358707 * 10–38 bis 0.17014118346 * 10+39
Genauigkeit: 56 Bit (16 Dezimalziffern)

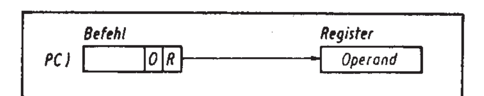


Bild 5 Registeradressierung

Operanden. Das nachgestellte Pluszeichen (+) zeigt an, daß die Adresse im Register nach der Operation erhöht wird.

Beispiele:
INC (2)+
CLR (R2)+

Autoinkrement-Indirekt-Adressierung (Modus 3)

Bei der Autoinkrement-Indirekt-Adressierung (Bild 8) wird der Inhalt des adressierten Registers als eine Adresse einer im Speicher befindlichen Adreßkonstanten der Operandenadresse interpretiert. Damit sind für den Operandenzugriff zwei Buszyklen erforderlich. Nach dem Zugriff zum Operanden wird der Inhalt des Registers immer um 2 erhöht, da das Register auf eine Adreßkonstante (2 Bytes) zeigt.
Das Register *ER* enthält die Adresse (Register-Indirekt) der Adresse (Speicher-Indirekt) des Operanden. Nach der Operation wird die Adresse im Register erhöht.

Beispiele:
INC @(2)+
CLR @(R1+1)+

Autodekrement-Adressierung (Modus 4)

Bei der Autodekrement-Adressierung (Bild 9) wird der Inhalt des adressierten Registers um die Länge des Operanden verringert (bei Byteverarbeitungsbefehlen um 1, bei Wortverarbeitungsbefehlen um 2). Bei den Registern 6 und 7 erfolgt immer eine Verringerung um 2. Anschließend wird der verringerte Inhalt des Registers als Adresse des Operanden interpretiert. Für den Zugriff zum Operanden wird ein Buszyklus benötigt. Das Registers *ER* enthält die Adresse des Operanden. Das vorgestellte Minuszeichen (−) zeigt an, daß die Adresse im Register vor der Operation erniedrigt wird.

Beispiele:
INC −(R2)
CLR −(%1+2)

Autodekrement-Indirekt-Adressierung (Modus 5)

Bei der Autodekrement-Indirekt-Adressierung (Bild 10) wird der Inhalt des adressierten Registers um 2 verringert. Anschließend wird der verringerte Inhalt des Registers als Adresse einer Adreßkonstanten der Adresse des Operanden interpretiert. Für den Zugriff zum Operanden werden zwei Buszyklen benötigt.
Das Register *ER* enthält die Adresse (Register-Indirekt) der Adresse (Speicher-Indirekt) des Operanden. Vor der Operation wird die Adresse im Register erniedrigt.

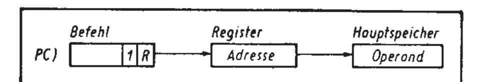


Bild 6 Register-Indirekt-Adressierung

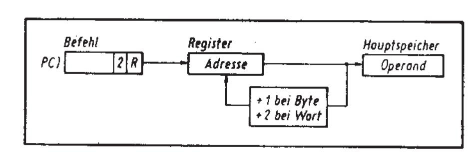


Bild 7 Autoinkrement-Adressierung

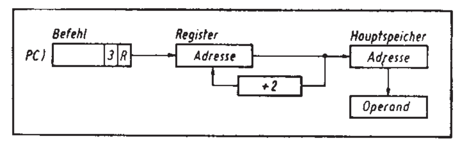


Bild 8 Autoinkrement-Indirekt-Adressierung

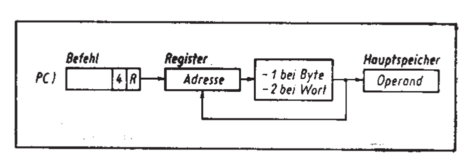


Bild 9 Autodekrement-Adressierung

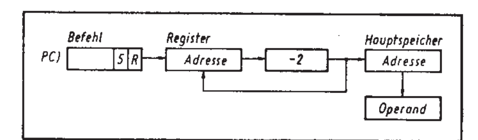


Bild 10 Autodekrement-Indirekt-Adressierung

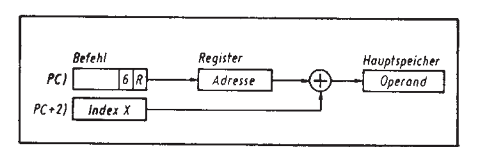


Bild 11 Index-Adressierung

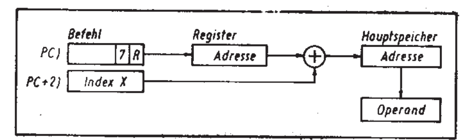


Bild 12 Index-Indirekt-Adressierung

Beispiele:
INC @−(R2)
CLR @−(2+1)

Index-Adressierung (Modus 6)

Bei der Index-Adressierung (Bild 11) wird zum Inhalt des adressierten Registers eine Indexkonstante addiert. Die Summe wird als die Adresse des Operanden interpretiert.
Die Indexkonstante belegt das nächste Wort nach dem Befehl. Der Befehlszähler (**PC**) wird nach dem Lesen der Indexkonstanten um 2 erhöht. Für den Zugriff zum Operanden werden zwei Buszyklen benötigt.
Der Ausdruck *E*, mit dem Inhalt des Registers *ER* addiert, ergibt die Adresse des Operanden. Der Wert des Ausdrucks *E* wird nach dem Befehlswort gespeichert. Der Ausdruck kann programmtechnisch die Basisadresse oder die Indexkonstante spezifizieren.

Beispiele:

```
INC FELD(R2); IM BEREICH FELD WIRD
            ; DAS DURCH R2 SPEZIFI-
            ; ZIERTE
            ; WORT INKREMENTIERT

CLR 20(R3)  ; LOESCHEN DES WORTES
            ; MIT DER VERSCHIEBUNG 20
            ; IM BEREICH, DER DURCH
            ; R3 SPEZIFIZIERT WIRD.
```

Index-Indirekt-Adressierung (Modus 7)

Bei der Index-Indirekt-Adressierung (Bild 12) wird der Inhalt des adressierten Registers mit einer Indexkonstanten addiert. Die Summe wird als die Adresse einer Adreßkonstanten der Adresse des Operanden interpretiert. Die Indexkonstante belegt das nächste Wort nach dem Befehl. Nach dem Lesen der Indexkonstanten wird der Befehlszähler (**PC**) um 2 erhöht. Für den Zugriff zum Operanden werden drei Buszyklen benötigt. Der Ausdruck *E*, mit dem Inhalt des Registers *ER* addiert, ergibt die Adresse der Adresse des Operanden. Der Wert des Ausdrucks *E* wird nach dem Befehlswort gespeichert.

Beispiele:
INC @FELD(R2)
CLR @10.(R1+2)

Anmerkungen zu Byteverarbeitungsbefehlen:

1. Bei den Adressierungsmodifikationen mit Inkrementierung bzw. Dekrementierung des Registerinhaltes (2 und 4) erfolgt bei den Byteverarbeitungsbefehlen eine Erhöhung bzw. Erniedrigung der Adresse um 1 und bei den Wortverarbeitungsbefehlen um 2.
2. Bei den indirekten Adressierungsmodifikationen mit Inkrementierung bzw. Dekrementierung des Registerinhaltes (3 und 5) erfolgt bei den Byte- und Wortverarbeitungsbefehlen eine Erhöhung bzw. Erniedrigung der Adresse um 2, da die Register auf die Adressen der Operanden verweisen.
3. Byteverarbeitungsbefehle in Verbindung mit dem Stackpointer des Systemstacks (**SP**) und dem Befehlszähler (**PC**) erhöhen bzw. erniedrigen diese Register immer um 2.
4. Byteverarbeitungsbefehle adressieren bei der Registeradressierung das niederwertige Byte im Register. Bei Byteverarbeitungsbefehlen in Verbindung mit dem **SP** bzw. **PC** wird ebenfalls immer das niederwertige Byte eines Wortes adressiert.
5. Beim Laden eines Bytes in ein Register (**MOVB**) wird das Byte in den niederwertigen Teil des Registers geladen, und der höherwertige Teil des Registers wird mit dem Vorzeichen des Bytes gefüllt (Signextension). Damit wird eine ganze Zahl aus dem Byteformat in das Wortformat überführt.

5.3.2. Die erweiterten Adressierungsarten
Direktwertadressierung (Modus 2 mit PC)

Der Wert des Operanden wird als sogenannter Direktwert (Bild 13) nach dem Befehlswort gespeichert. Da nach dem Lesen des Be-

fehlswortes der **PC** auf das nächste Wort zeigt, kann unter Verwendung des **PC** als Adressierungsregister der Direktwert gelesen werden. Bei Verwendung des Modus 2 wird der **PC** automatisch um 2 erhöht, so daß er nach Ausführung des Befehls mit Direktwertadressierung auf den nächsten Befehl zeigt.
Die Direktwertadressierung wird vor allem bei Zwei-Adreß-Befehlen für die Angabe des Quelloperanden verwendet, wenn er z. B. eine Konstante oder Maske darstellt. Der Assembler erkennt Direktwerte am vorangestellten Doppelkreuz „#".
Der Ausdruck *E* ist der Operand selbst. Der Wert des Ausdrucks wird nach dem Befehlswort als Direktwert gespeichert.

Beispiele:
```
MOV #FELD,R2       ; DIE ADRESSE FELD
                   ; WIRD GELADEN
ADD #^B1011,R2     ; ZUM INHALT DES
                   ; REGISTERS R1
                   ; WIRD DER WERT
                   ; 1011(2) ADDIERT
```

Absolute Adressierung (Modus 3 mit PC)

Die absolute Adresse des Operanden wird als sogenannter Direktwert (Bild 14) nach dem Befehlswort gespeichert. Da nach dem Lesen des Befehlswortes der **PC** auf den Direktwert zeigt, kann unter Verwendung des **PC** als Adressierungsregister und des Modus 3 zum Operanden zugegriffen werden. Dabei wird gleichzeitig der **PC** um 2 erhöht, so daß der **PC** nach Ausführung des Befehls auf den nächsten Befehl zeigt. Die absolute Adressierung soll nur dort eingesetzt werden, wo die Adresse absolut feststeht, z. B. zur Adressierung des Prozessorstatuswortes, der Geräteregister und Interruptvektoren. Der Assembler erkennt die absolute Adressierung an dem vorangestellten Sonderzeichen „@#".
Der Ausdruck *E* gibt die Adresse des Operanden an, die nach dem Befehlswort gespeichert wird. Der Ausdruck darf je nach Charakter des PA absolut oder relativ sein. Bei relativen (verschieblichen) Adressen erfolgt vom Taskbuilder die Umrechnung in eine absolute Adresse. Zur Programmlaufzeit steht somit hinter dem Befehlswort immer die absolute Adresse.

Beispiele:
```
MOV @#177776,R0; LADEN
               ; DES PROZESSOR-
               ; STATUS
CLR @#ALPHA
```

Relative Adressierung (Modus 6 mit PC)

Da sich bei der Index-Adressierung die Operandenadresse aus der Summe des Inhaltes des Adressierungsregisters und der Indexkonstanten ergibt, muß bei Verwendung des **PC** als Adressierungsregister die Indexkonstante die Differenz zwischen Operandenadresse und dem aktuellen Inhalt des **PC** betragen (Bild 15). Diese Differenz wird als relative Adresse bezeichnet. Diese Adressierungsart ist die am meisten verwendete. Der Assembler übersetzt alle Operandenangaben ohne besonderes Kennzeichen als relative Adressen.
Der Ausdruck *E* gibt je nach Charakter des PA die absolute oder relative (verschiebliche) Adresse des Operanden an. Als relative Adresse im Sinne der relativen Adressierung wird die Differenz zwischen dem Wert des Ausdrucks und der Adresse **PC** nach dem Lesen der Indexkonstanten des Befehls berechnet und als Indexkonstante nach dem Befehlswort gespeichert. Ist die Berechnung der relativen Adresse zur Assemblerzeit nicht möglich, so wird sie vom Taskbuilder ausgeführt. Zur Programmlaufzeit steht somit immer die relative Adresse des Operanden zum aktuellen **PC**-Inhalt nach dem Lesen der relativen Adresse im Wort der Indexkonstanten. Diese Adressierungsart ist für die Adressierung von symbolischen Speicherbereichen die am meisten genutzte.

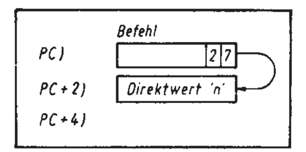


Bild 13 Direktwertadressierung

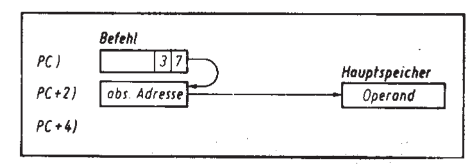


Bild 14 Absolute Adressierung

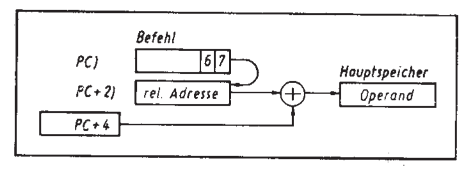


Bild 15 Relative Adressierung

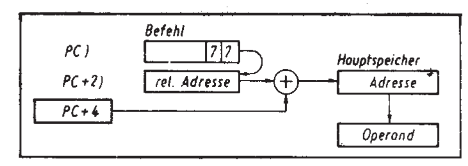


Bild 16 Indirekte relative Adressierung

Beispiele:
```
MOV ALPHA, BETA
CLR A+B-C
```

Indirekte relative Adressierung (Modus 7 mit PC)

Aus der Summe der relativen Adresse, die nach dem Befehlswort gespeichert ist, und dem Inhalt des PC ergibt sich die absolute Adresse einer Adreßkonstanten des Operanden (Bild 16). Der Assembler erkennt diese Adressierungsart am vorangestellten Sonderzeichen „@".
Der Ausdruck *E* gibt je nach Charakter des PA die absolute oder relative (verschiebliche) Adresse der Adresse des Operanden an. Weitere Bemerkungen siehe unter „Relative Adressierung".

Beispiele:
```
MOV @ADR, @BETA
CLR @A+2*<C-D>
```

5.3.3. Relative Adressierung in den Verzweigungsbefehlen

Der unbedingte und die bedingten Verzweigungsbefehle sind 1-Wort-Befehle, die im niederwertigen Byte eine relative Adresse (Verschiebung) enthalten. Diese Verschiebung (*V*) gibt die Anzahl der Worte an, um die vorwärts (+) oder rückwärts (−) gesprungen werden soll. Die Ausdrucksberechnung erfolgt wie bei der relativen Adressierung – mit dem Unterschied, daß anschließend der Wert durch 2 dividiert (Wortadressierung!) und auf 8 Bit verkürzt wird:
V = <*E* − . − 2>/2
Wenn *V* mit Vorzeichen größer als 8 Bit wird, so wird ein A-Fehler angezeigt.
Beispiele:
Folgende unbedingte Verzweigungsbefehle stehen auf der Adresse 120274(8):
a) in binärer Darstellung:
0 000 000 1   00 010 100
Operationskode   Verschiebung
in oktaler Darstellung: **000424**
Dieser Befehl bewirkt zur Programmlaufzeit eine Verzweigung nach:
PC+2∗Verschiebung=
120276+2∗(+24)=120346
b) in binärer Darstellung:
0 000 000 1 11 111 100
in oktaler Darstellung: **000774**
Dieser Befehl bewirkt zur Programmlaufzeit eine Verzweigung nach:
PC+2∗Verschiebung=**120276+2∗(−4)=120266**
c) in binärer Darstellung:
0 000 000 1 11 111 111
in oktaler Darstellung: **000777**
Dieser Befehl bewirkt zur Programmlaufzeit eine Verzweigung nach:
PC+2∗Verschiebung=**120276+2∗(−1)=120274**
Sprung auf den Befehl selbst (endlose Schleife)!

Fortsetzung folgt!

Programmieren mit MACRO-SM

Teil V
Dr. Thomas Horn
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

5.4. Datenformate

Adressen

Adressen werden wie vorzeichenlose ganze Zahlen (natürliche Zahlen) im Wortformat (16 Bit) gespeichert und verarbeitet. Durch die 16-Bit-Adressen ist der virtuelle Adreßraum auf 64 KByte beschränkt.

Intergerzahlen

Integerzahlen sind ganze Zahlen (mit Vorzeichen) im
- Byteformat (B) – 8 Bit
- Wortformat (I) – 16 Bit
- Langwortformat (L) – 32 Bit.

Grundsätzlich stehen alle arithmetischen und logischen Operationen für Integer-Zahlen im Wortformat zur Verfügung. Mit Ausnahme der Befehle für die arithmetischen Grundoperationen stehen auch alle Befehle für das Byteformat zur Verfügung. Für das Langwortformat ist nur ein Verschiebebefehl realisiert. Im Befehlssatz des FPP sind aber Konvertierungsbefehle zur Umwandlung in das D-Format vorhanden, so daß das L-Format effektiv über den FPP verarbeitet werden kann.
Ein Überlauf bei der Verarbeitung von Integerzahlen wird durch das V-Bit im Prozessorstatuswort (**PS**) angezeigt.

Natürliche Zahlen

Natürliche Zahlen sind ganze Zahlen ohne Vorzeichen im Byte- oder Wortformat. Für die Verarbeitung stehen die gleichen arithmetischen und logischen Befehle wie bei den Integerzahlen zur Verfügung. Ein auftretender Übertrag wird im C-Bit des **PS** angezeigt.

Gleitkommazahlen

Gleitkommazahlen werden durch FPP im
- F-Format (32 Bit) und
- D-Format (64 Bit)

verarbeitet. Es stehen umfangreiche arithmetische und Konvertierungsbefehle zur Verfügung. Bei Anlagen mit FIS sind nur 4 Befehle für das F-Format realisiert.
Eine Übersicht über die internen Darstellungen der Datenformate des SKR ist auf Seite 112 (MP 4/88) abgebildet.

5.5. Der Basisbefehlssatz des SKR

Der Basisbefehlssatz des SKR stellt den Standardbefehlsvorrat dar. Bei der Erläuterung der Befehle werden folgende Abkürzungen benutzt:

Mnemonischer Befehl

D – Ausdruck für die Adresse des Zieloperanden
S – Ausdruck für die Adresse des Quelloperanden
R – Registerausdruck
N – numerischer Ausdruck

numerischer Operationskode

DD – Zieloperandenfeld (6 Bit)
SS – Quelloperandenfeld (6 Bit)
R – Registerangabe (3 Bit)
XXX – Verschiebung (8 Bit), −128 bis +127
N – numerischer Wert (3 Bit)
NN – numerischer Wert (6 Bit)
NNN – numerischer Wert (8 Bit)

Operationsbeschreibung

d – Zieloperand
s – Quelloperand
r – Registerinhalt
(...) – Inhalt von ...
:= – Ergibtzeichen
\+ – Addition
− – Subtraktion, Vorzeichen (Zweierkomplement)
∗ – Multiplikation
/ – Division
~ – logische Negation (Einerkomplement)
∧ – logisches UND
∨ – inklusives logisches ODER
∀ – exklusives logisches ODER

Zu jedem Befehl wird die Aktualisierung der Flagbits des Prozessorstatusregisters (**PS**) angegeben, wobei folgende Symbolik verwendet wird:

∗ – wird entsprechend dem Ergebnis gesetzt oder gelöscht
– – wird nicht beeinflußt
Ø – wird gelöscht
1 – wird gesetzt.

Das Setzen bzw. Löschen der Flagbits erfolgt, falls nicht anders angegeben, nach folgenden Grundregeln:

1. Das N- Bit (negtiv) wird dem Vorzeichenbit des Resultates entsprechend gesetzt (höchstwertiges Bit).
2. Das Z-Bit (zero) wird gesetzt, wenn alle Bits des Ergebnisses Null sind, anderenfalls wird es gelöscht.
3. Das V-Bit (overflow) wird bei einem arithmetischen Überlauf gesetzt (Verletzung des Zahlenbereiches der ganzen Zahlen).
4. Das C-Bit (carry) wird bei einem Übertrag gesetzt (Verletzung des Zahlenbereiches der natürlichen Zahlen).
Wenn zu einem Wortverarbeitungsbefehl ein äquivalenter Byteverarbeitungsbefehl realisiert ist, so erfolgt unter Angabe der Mnemonik beider Befehle die Abhandlung gemeinsam. Ein Byteverarbeitungsbefehl ist im mnemonischen Operationskode durch ein nachgestelltes „**B**" gekennzeichnet.

5.5.1. Allgemeine Befehle

CLR ***D*** **ØØ5Ø*DD*** *d*:=Ø **Löschen** **N Z V C**
CLRB ***D*** **1Ø5Ø*DD*** (Clear) **Ø 1 Ø Ø**
Der Zieloperand wird gelöscht.

COM ***D*** **ØØ51*DD*** *d*:=~*d* **Komplement** **∗∗Ø1**
COMB ***D*** **1Ø51*DD*** (Complement)
Der Zieloperand wird logisch komplementiert (Einerkomplement), d. h., jede Binärziffer wird negiert. Das C-Bit wird gesetzt!

INC ***D*** **ØØ52*DD*** *d*:=*d*+1 **Inkrement** **∗∗∗–**
INCB ***D*** **1Ø52*DD*** (Increment)
Zum Zieloperanden wird eine 1 addiert. Das C-Bit wird nicht verändert!

DEC ***D*** **ØØ54*DD*** *d*:=*d*−1 **Dekrement** **∗∗∗∗**
DECB ***D*** **1Ø54*DD*** (Decrement)
Vom Zieloperanden wird eine 1 subtrahiert. Das Bit wird nicht verändert!

NEG ***D*** **ØØ53*DD*** *d*:=−*d* **Negation** **∗∗∗∗**
NEGB ***D*** **1Ø53*DD*** (Negate)
Vom Zieloperanden wird das Zweierkomplement gebildet (Subtraktion von Ø). Das V-Bit wird gesetzt, wenn die größte negative Zahl negiert wird, da sie kein positives Äquivalent hat und die Operation nicht ausgeführt werden kann. Das C-Bit wird prinzipiell mit einer Ausnahme gesetzt, wenn der Wert Null negiert wird.

TST ***D*** **ØØ57*DD*** *d*−Ø **Test** **∗∗ØØ**
TSTB ***D*** **1Ø57*DD*** (Test)
Der Zieloperand wird getestet, wobei N- und Z-Bit entsprechend gesetzt oder gelöscht werden.

SWAB ***D*** **ØØØ3*DD*** **Vertauschen der Bytes** (Swap bytes) **∗∗ØØ**
Höher- und niederwertiges Byte des Zieloperanden werden miteinander vertauscht. Die Zieladresse muß eine Wortadresse sein. Das N-Bit wird gesetzt, wenn im Ergebnis das Vorzeichen des niederwertigen Bytes (Bit 7) Eins ist. Das Z-Bit wird gesetzt, wenn im Ergebnis das niederwertige Byte Null ist.

MOV ***S,D*** **Ø1*SSDD*** *d*:=*s* **Transport** **∗∗Ø–**
MOVB ***S,D*** **11*SSDD*** (Move)
Der Quelloperand wird auf die Zieladresse transportiert. Der Quelloperand und das C-Bit bleiben unverändert. Das N- und Z-Bit werden entsprechend dem transportierten Operanden gesetzt. Bei MOVB mit Registeradressierung für die Zieladresse ist zu beachten, daß das höherwertige Byte mit dem Vorzeichen des geladenen niederwertigen Bytes gefüllt wird (Sign extension).

CMP ***S,D*** **Ø2*SSDD*** *s*−*d* Vergleich **∗∗∗∗**
CMPB ***S,D*** **12*SSDD*** (Compare)
Vergleich des Quelloperanden mit dem Zieloperanden durch Subtraktion des Zieloperanden vom Quelloperanden. Die Operanden bleiben unverändert. Das V-Bit wird bei einem arithmetischen Überlauf gesetzt (siehe **SUB**-Befehl). Das C-Bit wird bei einem Übertrag vom höchstwertigen **Bit** gesetzt. Beim Vergleich von natürlichen Zahlen ist das C-Bit gesetzt, wenn der Quelloperand kleiner als der Zieloperand ist.

5.5.2. Arithmetische Befehle (Festkomma)

ADD *S,D* Ø6*SSDD* *d*:=*d*+*s* **Addition (Add)** **∗∗∗∗**
Der Quelloperand wird zum Zieloperanden addiert. Der Zieloperand wird dabei durch das Resultat überschrieben, der Quelloperand bleibt unverändert. Das C-Bit wird bei einem Übertrag vom höchstwertigen Bit gesetzt. Das V-Bit wird beim arithmetischen

Überlauf gesetzt (wenn beide Operanden das gleiche Vorzeichen haben und das Ergebnis das inverse Vorzeichen hat).

SUB ***S,D*** **16*SSDD*** *d:=d−s* **Subtraktion** (Subtract) ****

Der Quelloperand wird vom Zieloperanden subtrahiert. Das C-Bit wird bei einem Übertrag vom höchstwertigen Bit gesetzt. Das V-Bit wird bei arithmetischem Überlauf gesetzt (Minuend und Subtrahend haben verschiedene Vorzeichen und das Vorzeichen des Resultats ist mit dem Vorzeichen des Subtrahenden identisch.

ADC ***D*** **0055*DD*** *d:=d+C* **Addition**
ADCB ***D*** **1055*DD*** **des C-Bits** (Add carry) ****

Das C-Bit wird zum Zieloperanden addiert.

SBC ***D*** **0056*DD*** *d:=d−C* Subtrak-
SBCB ***D*** **1056*DD*** tion des C-Bits (Subtract carry) ****

Das C-Bit wird vom Zieloperanden subtrahiert.

5.5.3. Logische Befehle

BIT ***S,D*** **03*SSDD*** *s∧d Bit Testen* **0−
BITB ***S,D*** **13*SSDD*** (Bit test)

Verknüpft die beiden Operanden konjunktiv und setzt entsprechend N- und Z-Bit. Die Operanden werden nicht verändert. Gewöhnlich werden durch den Befehl mittels einer Maske ein oder mehrere Bits des Zieloperanden getestet.

BIC ***S,D*** **04*SSDD*** *d(:~S)∧d* **Bit** **0−
BICB ***S,D*** **14*SSDD*** **Löschen** (Bit clear)

Es werden im Zieloperanden die Bits auf Null gesetzt, die im Quelloperanden (Maske) mit Eins belegt sind (Konjunktion mit negiertem Quelloperand).

BIS ***S,D*** **05*SSDD*** *d:=s∨d* **Bit Setzen** **0−
BISB ***S,D*** **15*SSDD*** (Bit set)

Es werden im Zieloperanden die Bits auf Eins gesetzt, die im Quelloperanden mit Eins belegt sind (Disjunktion).

5.5.4. Verschiebebefehle

ROR ***D*** **0060*DD*** **Zyklische Rechts-** ****
RORB ***D*** **1060*DD*** **verschiebung** (Rotate right)

Alle Bits des Zieloperanden werden um 1 Bit nach rechts verschoben. Das C-Bit wird in das höchstwertige Bit und Bit 0 in das C-Bit eingetragen (Bild 17).
Das V-Bit wird wie folgt gesetzt: V = N∀C. Damit zeigt das V-Bit an, daß der neue Wert des C-Bits gegenüber dem vorhergehenden geändert ist.



Bild 17 Zyklische Rechtsverschiebung

ROL ***D*** **0061*DD*** **Zyklische Links-** ****
ROLB ***D*** **1061*DD*** **verschiebung** (Rotate left)

Alle Bits des Zieloperanden werden um 1 Bit nach links verschoben. Das C-Bit wird in das Bit 0 und das höchstwertige Bit in das C-Bit eingetragen. Das V-Bit wird wie beim **ROR**-Befehl gesetzt (Bild 18).

ASR ***D*** **0062*DD*** **Arithmetische** ****
ASRB ***D*** **1062*DD*** **Rechtsverschiebung (Arithmetic shift right)**



Bild 18 Zyklische Linksverschiebung

Alle Bits des Zieloperanden werden um 1 Bit nach rechts verschoben. Das höchstwertige Bit bleibt unverändert. Bit 0 wird in das C-bit geladen. Dieser Befehl entspricht einer Division durch 2. Das V-Bit wird wie beim ROR-Befehl gesetzt (Bild 19).



Bild 19 Arithmetische Rechtsverschiebung

Hinweis:
Bei einem Byteverarbeitungsbefehl mit einem Registeroperanden bleibt das höherwertige Byte unverändert (Ausnahme: **MOVB**).

ASL ***D*** **0063*DD*** **Arithmetische** ****
ASLB ***D*** **1063*DD*** **Linksverschiebung (Arithmetic shift left)**

Alle Bits des Zieloperanden werden um 1 Bit nach links verschoben. Das höchstwertige Bit wird in das C-Bit geladen. In das Bit 0 wird eine Null geladen. Solange das V-Bit nicht gesetzt wird, entspricht die Operation einer Multiplikation mit 2. Das V-Bit wird wie beim **ROR**-Befehl gesetzt und zeigt eine Vorzeichenänderung an (Bild 20).



Bild 20 Arithmetische Linksverschiebung

5.5.5. Verzweigungsbefehle

Alle Verzweigungsbefehle sind 1-Wort-Befehle, die im niederwertigen Byte eine Wortverschiebung *XXX* (relative Adresse gemäß 5.3.3.) enthalten. Der Sprungbereich ist somit relativ zum aktualisierten **PC** (alter **PC** plus 2) −128 Worte rückwärts und +127 Worte vorwärts. Die Wortverschiebung wird im Befehlskode mit *XXX* dargestellt (*XXX = D* − **PC**).

Allgemeine Verzweigungsbefehle

BR ***D*** **000400+*XXX*** unbedingt (Branch)
BNE ***D*** **001000+*XXX*** bei ungleich Null (Branch if not equal) **Z=0**
BEQ ***D*** **001400+*XXX*** bei gleich Null (Branch if equal) **Z=1**
BPL ***D*** **100000+*XXX*** bei Plus (Branch if plus) **N=0**
BMI ***D*** **100400+*XXX*** bei Minus (Branch if minus) **N=1**
BVC ***D*** **102000+*XXX*** bei kein Überlauf (Br. if overflow clear) **V=0**
BVS ***D*** **102400+*XXX*** bei Überlauf (Br. if overflow is set) **V=1**
BCC ***D*** **103000+*XXX*** bei kein Übertrag (Br. if carry is clear **C=0**
BCS ***D*** **103400+*XXX*** bei Übertrag (Br. if carry is set) **C=1**

Hinweis:
Die Anwendung von **BPL/BMI** nach dem **CMP/CMPB**-Befehl und anderen arithmetischen Befehlen führt bei Überschreitung des Wertebereiches zu Fehlern, da das C- bzw. V-Bit vom Verzweigungsbefehl nicht analysiert wird. In Abhängigkeit davon, ob die 16-Bit-Zahlen als ganze Zahlen oder natürliche Zahlen interpretiert werden, sollten die Verzweigungsbefehle der nachfolgenden Gruppen Anwendung finden.

Verzweigungsbefehle nach Operationen mit ganzen Zahlen

BGE ***D*** **002000+*XXX*** bei größer oder gleich Null (Branch if greater than or equal) **N∀V=0**
BLT ***D*** **002400+*XXX*** bei kleiner als Null (Branch if less than) **N∀V=1**
BGT ***D*** **003000+*XXX*** bei größer als Null (Branch if greater than) **Z∨(N∀V)=30**
BLE ***D*** **003400+*XXX*** bei kleiner oder gleich Null (Branch if less than or equal **Z∨(n∀V)=1**

Diese Befehle werden immer nach arithmetischen Operationen mit ganzen Zahlen angewendet, da sonst nach einer möglichen Überschreitung des Wertebereiches falsch verzweigt wird.

Verzweigungsbefehle nach Operationen mit natürlichen Zahlen

BHI ***D*** **101000+*XXX*** Bei größer (Branch if higher) **C∨Z=0**
BLOS ***D*** **101400+*XXX*** bei kleiner oder gleich (Branch if lower or same) **C∨Z=1**
BHIS ***D*** **103000+*XXX*** bei größer oder gleich (Branch if higher or same) **C=0**
BLO ***D*** **103400+*XXX*** bei kleiner (Branch if lower) **C=1**

Diese Befehle werden nach arithmetischen Operationen mit natürlichen Zahlen (Adressen) angewendet, da sonst nach einer möglichen Überschreitung des Wertebereichs falsch verzweigt wird.

5.5.6. Sprungbefehle und Befehle zur Unterprogrammorganisation

JMP ***D*** **0001*DD*** Unbedingter Sprung (Jump)

Der **JMP**-Befehl gestattet einen Sprung zu einer beliebigen Adresse, da außer der Registeradressierung alle Adressierungsmodifikationen zulässig sind. Die Zieladresse muß immer geradzahlig sein. Die Adressierungsmodikation 0 bewirkt einen Interrupt nach Adresse 000010 (Verbotener Befehl) und eine ungerade Adresse 000004 (Time out / Verbotene Adresse). Die Flags des PS werden nicht verändert.

Operation: **PC**:=*d*

JSR *R,D* 004*RDD* Unterprogrammsprung (unbedingt) (Jump to subroutine)

Der **JSR**-Befehl gestattet einen Sprung zu einem Unterprogramm mit einer bestimmten Anfangsadresse. Vor dem Sprung wird der alte PC-Inhalt in das Register R gerettet, um eine Rückkehr zum Hauptprogramm zu ermöglichen. Damit der Inhalt des Registers R nicht verloren geht, wird er zuvor in den Systemstack gerettet. Der Sprung wird dann analog dem **JMP**-Befehl ausgeführt.

Operation:

tmp:=*d* (Die Zieladresse wird berechnet und in ein internes temporäres Register gespeichert)
−(SP):=*R* (Inhalt von R wird gerettet)
R:=**PC** (**PC** wird nach R gerettet)
PC:=*tmp* (Sprung wird ausgeführt)

RTS *R* 00020*R* Rücksprung aus dem Unterprogramm

Vom Register *R* wird die Rücksprungadresse geladen. Der Inhalt des Registers *R* wird vom Stack geladen. Die Flags des **PS** werden nicht beeinflußt.

Operation: **PC**:=*R* (Laden der Rücksprungadresse von *R*)
R:=**(SP)+** (Laden des alten Registerinhaltes vom Stack)

5.5.7. Steuerbefehle

HALT 000000 Halt

Der Prozessor geht in den statischen Haltzustand. Die Datenanzeige des Bedienpultes gibt den Inhalt des Registers **R0** an. Die Adreßanzeige gibt die Adresse des nächsten Befehls an. Durch Drücken der Fortsetzungstaste (продолжать/Continue) kann das Programm fortgesetzt werden. Im Nutzer-Modus ruft der Befehl eine Unterbrechung hervor.

WAIT 000001 Warten auf Interrupt (Wait for interrupt)

Der Prozessor geht in den dynamischen Stop und erwartet ein Interrupt. Der **PC** zeigt auf den nächsten Befehl, so daß nach der Interruptbehandlungsroutine (**RTI**-Befehl) der nächste Befehl nach **WAIT** ausgeführt werden kann. Im Nutzermodus wird der Befehl als **NOP**-Befehl ausgeführt.

RESET 000005 Rücksetzen des externen Busses (Reset external Bus)

Der Prozessor sendet auf den Einheitsbus für 10 ms das Signal INIT. Dadurch werden alle Geräte in ihren Grundzustand rückgesetzt. Im Nutzermodus wird der Befehl als **NOP**-Befehl ausgeführt.

RTI 000002 Rückkehr von der Interruptbehandlungsroutine (Return from Interrupt)

PC und **PS** werden vom Systemstack geladen. Es wird der Rücksprung in das unterbrochene Programm organisiert sowie der alte Flagzustand eingestellt.

Operationen:

PC:=(SP)+ (Laden der Rücksprungadresse aus dem Systemstack)
PS:=(SP)+ (Laden des alten PS aus dem Systemstack)

RTT 000006 Rückkehr von der Trapbehandlungsroutine (return from trap)

Der Befehl wird wie der RTI-Befehl ausgeführt. Der Unterschied ist, daß der **RTT**-Befehl den T-Bit-Trap blockiert, so daß es bei gesetztem T-Bit im **PS** erst nach dem nächsten Befehl wirksam wird. Beim **RTI**-Befehl würde der T-Bit-Trap sofort wirksam werden. Dieser Befehl ist ein Zusatzbefehl für Rechner mit Hauptspeicherverwaltungseinheit. Bei Rechenanlagen vom Typ SM3 wird der T-Bit-Trap auch beim **RTI**-Befehl erst nach einem Befehl wirksam.

NOP 000240 Keine Operation (No operation)
CLC 000241 Löschen des C-Bits (Clear C)
CLV 000242 Löschen des V-Bits (Clear V)
CLZ 000244 Löschen des Z-Bits (Clear Z)
CLN 000250 Löschen des N-Bits (Clear N)
CCC 000257 Löschen aller Flags (Clear all condition codes)
SEC 000261 Setzen des C-Bits (Set C)
SEV 000262 Setzen des V-Bits (Set V)
SEZ 000264 Setzen des Z-Bits (Set Z)
SEN 000270 Setzen des N-Bits (Set N)
SCC 000277 Setzen aller Flags (Set all condition codes)

Die letzten vier Bits der Operationskodes stimmen der Bedeutung nach mit den Flagbits des **PS** überein. Es ist deshalb möglich, numerische Operationskodes zum Löschen/Setzen von zwei bzw. drei Flags zu bilden, die vom Assembler nicht unterstützt werden, z. B.:

000256 Löschen vom **V**-, **Z**- und N-Bit

EMT N 104000+*NNN* Emulatortrap-Befehl
TRAP N 104400+*NNN* Trap-Befehl

Beide Befehle rufen eine synchrone Unterbrechung (Trap) hervor und werden gleich ausgeführt.

Operation:

−(SP):=PS (Retten des alten **PS**)
−(SP):=PC (Retten der Rücksprungadresse)
PC:=(*IV*) (Laden der Adresse der Trap-Behandlungsroutine vom Interruptvektor ***IV***)
PS:=(*IV*+2) (Laden des neuen PS vom Interruptvektor ***IV***+2)

Der Interruptvektor für den **EMT**-Befehl steht auf Adresse 30(8) und für den **TRAP**-Befehl auf Adresse 34(8). Der **EMT**-Befehl ist für die Systemsoftware reserviert und darf vom Anwenderprogrammierer nicht benutzt werden. Der **TRAP**-Befehl steht für die allgemeine Verwendung zur Verfügung.

BPT 000003 Trap-Befehl für Unterbrechungspunkte (Breakpoint trap)

Dem **EMT**- und **TRAP**-Befehl entsprechend ruft dieser Befehl eine synchrone Unterbrechung hervor. Der Interruptvektor für den **BPT**-Befehl steht auf der Adresse 14(8). Der T-Bit-Trap nutzt den gleichen Interruptvektor. Der **BPT**-Befehl und der T-Bit-Trap werden vom Testhilfesystem ODT des Betriebssystems genutzt.

IOT 000004 Trap-Befehl für die Ein- und Ausgabe (I/O-Trap)

Der **IOT**-Befehl wird dem **EMT**- und **TRAP**-Befehl entsprechend ausgeführt. Sein Interruptvektor steht auf der Adresse 20(8).
Er wurde vom E/A-Ruf der Lochbandsoftware genutzt.

MFPI *S* 0065*SS* Transport vom vorhergehenden Befehlsadreßraum (Move from previous Instruction Space) **0−

Der Befehl ist ein Zusatzbefehl bei Rechnern mit Hauptspeicherverwaltungseinheit. Er gestattet das Lesen von einzelnen Worten aus dem vorhergehenden Befehlsadreßraum. Das gelesene Wort wird in den Stack geschrieben.

Operation:
tmp :=*S*
−(SP) :=*tmp*

Die Befehls- und die Stackadresse werden entsprechend dem aktuellen Satz der Segmentadreßregister (Bit 14, 15 des **PS**) auf den physischen Speicher abgebildet, während für die Quelladresse der Segmentadreßregistersatz gemäß Bit 12, 13 des **PS** genutzt wird.

MTPI *D* 0066*DD* Transport in den vorhergehenden Befehlsadreßraum (Move to previous Instruction Space) **0−

Der Befehl ist die Umkehrung des **MFPI**-Befehls. Er gestattet das Schreiben von einzelnen Worten in den vorhergehenden Befehlsadreßraum. Das zu schreibende Wort wird aus dem Stack gelesen:

Operation:
tmp := **(SP)+**
D :=*tmp*

Die Befehls- und die Stackadresse werden entsprechend dem aktuellen Satz der Segmentadreßregister (Bit 14, 15 des PS) auf den physischen Speicher abgebildet, während für die Zieladresse der Segmentadreßregistersatz gemäß Bit 12, 13 des **PS** genutzt wird.

SPL *N* 00023*N* Setzen der Priorität (Set priority level) ––––

Der Befehl ist ein Zusatzbefehl beim K 1620 und bei der Elektronika-60. Die niederwertigen drei Bits des Befehls werden als Prozessorpriorität in das **PS** geladen. Im Nutzermodus wird der Befehl als **NOP**-Befehl ausgeführt.

MFPS *D* 1067*DD* Speichern des PS (Move byte from Processor status word) **0−

Der Befehl ist ein Zusatzbefehl des K 1620 und der Elektronika-60. Die niederwertigen 8 Bits des PS werden unter der Zieladresse abgespeichert. Die Zieladresse wird als Byteadresse interprtiert. Das Z-Bit wird gesetzt, wenn alle Bits im niederwertigen Byte des **PS** null sind.

MTPS *S* 1064*SS* Laden des PS (Move byte to processor status word) ****

Der Befehl ist ein Zusatzbefehl des K 1620 und der Elektronika-60. Die Bit 0–3 des Quelloperanden werden in das **PS** geladen. Die Quelloperandenadresse wird als Byte-

adresse interpretiert. Das T-Bit kann mit diesem Befehl nicht gesetzt werden.

5.6. Der erweiterte Befehlssatz des SKR

MUL ***S,R*** **070*RSS*** *(R,R*∨1):=*r*s* ∗∗0∗
Multiplikation (Multiply)

Es werden zwei ganze Zahlen (16-Bit) multipliziert, wobei ein 32-Bit-Produkt entsteht. Wenn *R* geradzahlig ist, wird im geradzahligen Register der höherwertige Teil des Produkts und im darauffolgenden ungeradzahligen Register der niederwertige Teil gespeichert. Wenn *R* ungeradzahlig ist, so wird nur der niederwertige Teil des Produkts gespeichert. Das C-Bit wird gesetzt, wenn das Ergebnis mehr als 15stellig wird (Das C-Bit muß, vorher gelöscht werden!).

Beispiele:

```
MOV ALPHA,R0  ;LADEN DES ERSTEN FAKTORS
MUL M0,R0     ;MULTIPLIZIEREN MIT K0
              ;ERGEBNIS IN R0,R1
CLC           ;LOESCHEN DES C-BITS
MOV FELD(R5),
           R3 ;LADEN DES
              ;ERSTEN FAKTORS
MUL FELD+2
      (R5),R3 ;MULTIPLIZIEREN
BCS FEHLER    ;----→ UEBERLAUFFEHLER
```

Hinweis:
Bei den Zusatzbefehlen **MUL**, **DIV**, **ASH** und **ASHC** ist die Operandenfolge im Assemblerformat gegenüber dem numerischen Maschinenbefehl vertauscht.

DIV ***S,R*** **071*RSS*** **(*R,R*+1):=(*R,R*+1)/s** ∗∗∗∗
Division (Divide)

Es wird ein 32stelliger Dividend (Ganze Zahl) durch einen 16stelligen Divisor dividiert. Der Quotient (16 Bit) wird im Register *R* und der Divisionsrest im Register *R*+1 gespeichert. Der Rest hat das gleiche Vorzeichen wie der Dividend. Das Register *R* muß immer geradzahlig sein. Das V-Bit wird gesetzt, wenn der Divisor 0 ist oder der Quotient größer als 15 Bit wird. Das C-Bit wird gesetzt, wenn durch 0 dividiert werden soll, anderenfalls wird es gelöscht.

Beispiele:

```
CLR R2          ;BEI NATUERLICHEN
                ;ZAHLEN ZULÄSSIG
MOV #5000.,R3   ;LADEN DES DIVIDENDEN
DIV K2,R2       ;DIVIDIEREN DURCH K2
..
MOV
     GAMMA,R3   ;LADEN DES
                ;16-BIT-DIVIDENDEN
                ;(GANZZAHLIG)
SXT R2          ;LADEN R2 MIT
                ;VORZEICHEN VON R3
DIV 10.,R2      ;DIVIDIEREN DURCH 10.
```

SXT ***D*** **0067*DD*** **Vorzeichen ausdehnung** (Sign extension) −∗0−

Bei N=0 wird der Zieloperand gelöscht. Bei N=1 wird der Zieloperand auf −1 gesetzt. Dieser Befehl wird insbesondere bei zur Umwandlung einer ganzen Zahl aus dem Wortformat in das Doppelwortformat benutzt (Beispiel siehe **DIV**-Befehl).

XOR ***R,D*** **074*RDD*** *d:=r*∀*d* **Exclusives ODER** (Exclusive OR) ∗∗0−

Der Registerinhalt und der Zieloperand werden nach der Funktion „Exklusiv-ODER" (Antivalenz) verknüpft und das Resultat unter der Zieladresse gespeichert.

Beispiel:

```
XOR    R2,SYM0
vorher:  (R2)=0123757 (SYM0)=0137744 1 0 1 1
nachher: (R2)=0123757 (SYM0)=0014013 0 0 0 1
```

ASH ***S,R*** **072*RSS*** **Arithmetische Verschiebung** (Arithmetic shift) ∗∗∗∗

Das Register *R* wird arithmetisch nach rechts oder links verschoben, wobei die Richtung und die Anzahl der Verschiebeschritte durch die 6 niederwertigen Bits des Quelloperanden festgelegt werden. Die artihmetische Verschiebung wird analog dem ASR/ASL-Befehl durchgeführt. Das C-Bit enthält immer das letzte herasusgeschobene Bit. Das V-Bit wird gesetzt, wenn während der Linksverschiebung ein Vorzeichenwechsel eintritt. Ein positiver Quelloperand bewirkt eine Linksverschiebung und ein negativer eine Rechtsverschiebung. Der Wertevorrat des Quelloperanden liegt im Bereich von −32. bis +31.

Beispiele:

```
ASH #4,R2      ;LINKSVERSCHIEBUNG
               ;UM 4 BIT
ASH SHIFT,R0   ;DIE VERSCHIEBUNG
               ;VON R0 WIRD DURCH DEN
               ;INHALT VON SHIFT DEFINIERT
```

ASHC ***S,R*** **073*RSS*** **Kombinierte arithmetische Verschiebung** (Arithmetic shift combined) ∗∗∗∗

Wenn *R* geradzahlig ist, wird das Doppelregister *R,R*+1 analog dem **ASH**-Befehl verschoben. Es erfolgt ein Übertrag zwischen Bit 15 von *R*+1 und Bit 0 von *R* und umgekehrt. (Weitere Hinweise siehe ASH-Befehl.) Wenn *R* ungeradzahlig ist, wird das ungeradzahlige Register entsprechend dem Wert des Quelloperanden zyklisch (über 16 Bit) verschoben; da ein direkter Übertrag zwischen Bit 0 und Bit 15 besteht. Das C-Bit enthält den letzten Übertrag zweischen beiden Bits. Das V-Bit wird bei Vorzeichenwechsel gesetzt.

Beispiel:

```
ASHC #-5,R4    ;(R4,R5) WERDEN UM 5 BIT
               ;NACH RECHTS VERSCHOBEN
               ;DIVISION DURCH 32
ASCH (R2)+,R1  ;R1 WIRD ZYKLISCH VER-
               ;SCHOBEN (R2)+ LEGT RICH-
               ;TUNG UND ANZAHL FEST
```

MARK ***N*** **0064*NN*** **Unterprogramm-Rücksprungunterstützung** (Mark) −−−−

Dieser Befehl wird durch die Standardunterprogramme benutzt. Er gestattet die Säuberung des Stacks und hilft bei der Organisation des Rücksprungs. Der Wert *NN* im Befehlskode gibt die Anzahl der im Stack zu löschenden Parameter an.

Operation:

SP:=PC+2∗*NN* (Laden der Adresse des alten Inhaltes des Registers **R5)**
PC:=R5 (Laden der Rücksprungadresse aus **R5)**
R5:=(SP)+ (Laden des **R5** mit dem alten Inhalt)

N Z V C

Beispiele:

```
        MOV R5,-(SP)       ;DEN INHALT
                           ;VON R5 RETTEN
        MOV P1,-(SP)       ;LADEN VON N
                           ;PARAMETER FUER DAS
        MOV P2,-(SP)       ;UNTERPROGRAMM
                           ;IN DEN STACK
        ...
        MOV PN,-(SP)
        MOV AMRKN,-(SP)    ;LADEN DES BEFEHLS
                           ;MARK N IN DEN STACK
        MOV SP,R5          ;LADEN DER ADRESSE
                           ;DES MARK-BEFEHLS
        ...
        JSR PC,UPN         ;AUSFUEHREN DES UP
AMRKN:  MARK N             ;DEFINITION DES
                           ;MARK-BEFEHLS
        ...
UPN:    ...                ;ANFANG DES UP
        ...
        RTS R5             ;RUECKSPRUNG INS
                           ;HAUPTPROGRAMM
```

Bei dem **RTS-R5**-Befehl erfolgt der Sprung in den Stack zur Ausführung des **MARK**-Befehls. Dabei wird in **R5** aus dem Stack die Rücksprungsadresse geladen. Beim **MARK**-Befehl erfolgen dann die Korrektur des **SP** auf die Adresse vor der Rücksprungadresse und das Wiederherstellen des alten Inhaltes von **R5**.

SOB ***R,D*** **077*RNN*** **Zyklusbefehl** (Subtract one and branch) N Z V C

Mit dem **SOB**-Befehl lassen sich kleinere Schleifen organisieren. Vom Register *R* wird eine Eins subtrahiert. Wenn das Ergebnis ungleich Null ist, ist die Schleife noch nicht beendet und die Verschiebung wird doppelt vom **PC** subtrahiert (Wortverschiebung!). Es ist somit nur ein Rückwärtssprung um max. 63 Worte möglich.

Beispiel:

```
      MOV #20.,R1   ;20. DURCHLAEUFE
2$:   ...           ;BEGINN DER SCHLEIFE
      ...
      SOB R1,2$     ;TEST AUF SCHLEIFEN-
                    ;ENDE
```

Der vorliegende MP-Kurs versuchte eine Einführung in die Makroassemblersprache MACRO-SM für die Klein- und Mikrorechner des SKR zu geben. Im Abschnitt 5 wurde vor allem auf die Adressierungsarten sowie die Basisbefehle und den erweiterten Befehlssatz eingegangen, die von den meisten Mikroprozessorschaltkreisfamilien realisiert werden. Nicht eingegangen werden konnte auf die 46 Befehle des Gleitkommaprozessors, über den die meisten Kleinrechenanlagen verfügen. Hierzu sei auf die einschlägige Fachliteraturr (Teil I) und die Dokumentationen verwiesen. Auch mußte leider aus Platzgründen auf das eingangs angekündigte abschließende Programmierbeispiel verzichtet werden.

Schluß